Cross-Layer Cloud Performance Monitoring, Analysis and Recovery

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
vorgelegt von

M.Sc. Afef Mdhaffar
geboren in Sfax, Tunesien

Marburg, 2014
Hochschulkennziffer 1180

Vom Fachbereich Mathematik und Informatik der
Philipps-Universität Marburg als Dissertation am
19.12.2014
angenommen.

Erstgutachter: Prof. Dr. Bernd Freisleben, Philipps-Universität Marburg

Zweitgutachter: Prof. Dr. Mohamed Jmaiel, Université de Sfax, Tunisia

Tag der Einreichung: 20.11.2014
Tag der mündlichen Prüfung: 19.12.2014

Abstract

The basic idea of Cloud computing is to offer software and hardware resources as services. These services are provided at different layers: Software (Software as a Service: SaaS), Platform (Platform as a Service: PaaS) and Infrastructure (Infrastructure as a Service: IaaS). In such a complex environment, performance issues are quite likely and rather the norm than the exception. Consequently, performance-related problems may frequently occur at all layers. Thus, it is necessary to monitor all Cloud layers and analyze their performance parameters to detect and rectify related problems.

This thesis presents a novel cross-layer reactive performance monitoring approach for Cloud computing environments, based on the methodology of Complex Event Processing (CEP). The proposed approach is called CEP4Cloud. It analyzes monitored events to detect performance-related problems and performs actions to fix them. The proposal is based on the use of (1) a novel multi-layer monitoring approach, (2) a new cross-layer analysis approach and (3) a novel recovery approach.

The proposed monitoring approach operates at all Cloud layers, while collecting related parameters. It makes use of existing monitoring tools and a new monitoring approach for Cloud services at the SaaS layer. The proposed SaaS monitoring approach is called AOP4CSM. It is based on aspect-oriented programming and monitors quality-of-service parameters of the SaaS layer in a non-invasive manner. AOP4CSM neither modifies the server implementation nor the client implementation.

The defined cross-layer analysis approach is called D-CEP4CMA. It is based on the methodology of Complex Event Processing (CEP). Instead of having to manually specify continuous queries on monitored event streams, CEP queries are derived from analyzing the correlations between monitored metrics across multiple Cloud layers. The results of the correlation analysis allow us to reduce the number of monitored parameters and enable us to perform a root cause analysis to identify the causes of performance-related problems. The derived analysis rules are implemented as queries in a CEP engine. D-CEP4CMA is designed to dynamically switch between different centralized and distributed CEP architectures depending on the load/memory of the CEP machine and network traffic conditions in the observed Cloud environment.

The proposed recovery approach is based on a novel action manager frame-

work. It applies recovery actions at all Cloud layers. The novel action manager framework assigns a set of repair actions to each performance-related problem and checks the success of the applied action.

The results of several experiments illustrate the merits of the reactive performance monitoring approach and its main components (i.e., monitoring, analysis and recovery). First, experimental results show the efficiency of AOP4CSM (very low overhead). Second, obtained results demonstrate the benefits of the analysis approach in terms of precision and recall compared to threshold-based methods. They also show the accuracy of the analysis approach in identifying the causes of performance-related problems. Furthermore, experiments illustrate the efficiency of D-CEP4CMA and its performance in terms of precision and recall compared to centralized and distributed CEP architectures. Moreover, experimental results indicate that the time needed to fix a performance-related problem is reasonably short. They also show that the CPU overhead of using CEP4Cloud is negligible. Finally, experimental results demonstrate the merits of CEP4Cloud in terms of speeding up the repair and reducing the number of triggered alarms compared to baseline methods.

Zusammenfassung

Die Grundidee des Cloud Computing ist es, Software und Hardware-Ressourcen als Dienste anzubieten. Diese Dienste werden in verschiedenen Schichten bereitgestellt, als Software (Software as Service: SaaS), Plattform (Platform as a Service: PaaS) und als Infrastruktur (Infrastructure as a Service: IaaS). In diesem komplexen Umfeld stellt eine gute Koordination eine besondere Herausforderung dar, insbesondere, weil Leistungseinbußen oft in jeder Schicht zu verzeichnen sind. Daher ist es notwendig, Leistungsparameter aller Schichten des Cloud-Systems zu überwachen, um mögliche Probleme frühzeitig zu erkennen, zu analysieren und zu beheben.

Diese Arbeit stellt einen neuen, Schichten übergreifenden Ansatz zur Überwachung und Steuerung von Cloud-Computing-Umgebungen vor. Dieser basiert auf dem sogenannten Complex Event Processing (CEP), also der Verarbeitung komplexer Ereignisse. Der vorgeschlagene Ansatz wird als CEP4Cloud bezeichnet. Er analysiert die überwachten Ereignisse, um leistungsbezogene Probleme zu erkennen und leitet auch Maßnahmen zu ihrer Behebung ein. Der Vorschlag basiert auf der Verwendung eines (1) neuartigen Mehrschichtenüberwachungskonzepts, (2) einer speziellen Schichten übergreifenden Analyse und (3) einem neuen Reparaturverfahren.

Der vorgeschlagene Überwachungsentwurf berücksichtigt alle Schichten des Cloud-Systems bei der Erhebung der notwendigen Parameter. Er nutzt bereits vorhandene Überwachungswerkzeuge zusammen mit einem neuen Überwachungskonzept für Dienste in der SaaS-Schicht. Dieser Ansatz wird als AOP4CSM bezeichnet. Mittels aspektorientierter Programmierung werden Qualitätsparameter transparent aus der SaaS-Schicht ermittelt. Dabei ändert AOP4CSM weder die server-seitige Implementation noch die Software des Klienten.

Der erarbeitete Schichten übergreifende Analyseansatz basiert auf der CEP-Methodik und wird als D-CEP4CMA bezeichnet. Anstatt einer aufwändigen manuellen Spezifikation von Abfragen an die kontinuierlich überwachten Ereignisströme werden CEP-Abfragen hierbei automatisch aus Korrelationen zwischen den erfassten Metriken der verschiedenen Cloud-Schichten abgeleitet. Die Korrelationsanalyse erlaubt eine Reduktion der Anzahl überwachter Parameter und mit ihnen auch eine effektive Identifikation der Ursachen leistungsbezogener Probleme. Die abgeleiteten Analyseregeln werden als Abfragen in einem CEP-Modul realisiert. D-CEP4CMA wurde so entworfen, dass abhängig von der Netzwerklast

und den Speicherresourcen des CEP-Rechners dynamisch zwischen verschiedenen zentralen und verteilten CEP-Architekturen in der Cloud umgeschaltet werden kann.

Der vorgeschlagene Wiederherstellungs-Ansatz basiert auf einem neuartigen Verfahren für die Verwaltung von Reparatur-Aktivitäten in den drei Cloud-Schichten. Ein spezieller Aktionsmanager bringt, je nach der Art des identifizierten Problems, verschiedene Reparaturmaßahmen zur Anwendung und überprüft auch deren Erfolg.

Die durchgeführten Experimente veranschaulichen die Vorteile der vorgeschlagenen reaktiven Cloud-Monitoring und -Steuerungslösung in den Bereichen Monitoring, Analyse und Reparatur. Einerseits zeigen die Experimente eine hohe Effizienz von AOP4CSM im Hinblick auf geringe zusätzliche Laufzeiten und Speicherplatzbedarf. Zugleich werden im Vergleich zu einfachen schwellwertbasierten Methoden gute Ergebnisse bei der Identifikation von leistungsbezogenen Problemen erzielt (im Sinne der statistischen Größen precision und recall). Dasselbe gilt auch im Hinblick auf das gute Abschneiden von D-CEP4CMA im Vergleich zu zentralisierten und verteilten CEP-Architekturen. Die experimentellen Ergebnisse zeigen auch, dass mit der vorgeschlagenen Methode nur relativ wenig Zeit benötigt wird, um leistungsbezogene Problem zu beheben. Die erzeugte CPU-Last durch Verwendung von CEP4Cloud ist dabei vernachlässigbar. Im Vergleich zu anderen Methoden liefert CEP4Cloud also schnellere Reparaturmaßnahmen bei einer geringeren Anzahl ausgelöster Fehlalarme.

Résumé

L'idée fondamentale du Cloud computing consiste à tout offrir (i.e. ressources matérielles et logicielles) comme un service. Les services du Cloud sont fournis à différentes couches : Software (Software as a Service : SaaS), Plateforme (Platform as a Service : PaaS) et Infrastructure (Infrastructure as a Service : IaaS). Dans de tels environnements aussi complexes, les dégradations de performance sont très fréquentes et peuvent toucher toutes les couches du Cloud. Ainsi, il est nécessaire de surveiller toutes ces couches et d'analyser leurs paramètres de performance, pour résoudre les problèmes les concernant.

Cette thèse présente une nouvelle approche de « monitoring » pour contrôler les environnements du Cloud computing. L'approche proposée est réactive et inter-couches. Elle est basée sur la méthodologie de Complex Event Processing (CEP), et ainsi appelée CEP4Cloud. Dans cette approche, les évènements de monitoring sont analysés pour détecter les dégradations de performance et lancer les actions de réparation nécessaires au rétablissement de l'état du Cloud. CEP4Cloud est basée sur l'utilisation de trois nouvelles méthodes: (1) une approche multi-couches de monitoring, (2) une approche inter-couches d'analyse et (3) une approche de réparation.

L'approche de monitoring proposée permet de surveiller toutes les couches du Cloud en collectant les paramètres de performance qui y sont liés. Elle est composée des outils de monitoring existants et de notre nouvelle approche de monitoring, appelée AOP4CSM. AOP4CSM permet de surveiller la couche software. Elle est basée sur la programmation orientée aspect et permet de collecter les paramètres de la qualité de service de la couche Software, sans modifier ni l'implémentation du service, ni celle du client.

L'approche inter-couches d'analyse proposée est appelée D-CEP4CMA. Elle est basée sur la méthodologie de Complex Event Processing (CEP). Les requêtes utilisées par le moteur de CEP sont déduites d'une étude de corrélations entre les différents paramètres du Cloud, au lieu d'être spécifiées manuellement. Les résultats de notre analyse de corrélations nous ont permis de réduire le nombre de paramètres à surveiller. En outre, cette analyse nous a été utile pour identifier la cause d'une dégradation de performance, tout en adoptant la méthode « Root Cause Analysis ». Les règles d'analyse déduites sont ainsi implémentées, en tant que requêtes au niveau du moteur CEP. D-CEP4CMA est capable de sélectionner dynamiquement, au moment de l'exécution, l'architecture la plus adaptée (cen-

tralisée ou distribuée) à l'état du Cloud. Le processus de sélection est basé sur la charge et la mémoire de la machine hébergeant le moteur CEP, ainsi que sur l'état du Cloud.

L'approche de réparation proposée est basée sur l'utilisation d'un nouveau framework de gestion d'actions. Ce framework applique des actions de réparation sur toutes les couches du Cloud. En outre, il attribue plusieurs actions de réparation à toute dégradation de performance et vérifie le succès de l'action appliquée.

Les résultats de plusieurs expérimentations ont montré les avantages de notre approche réactive de monitoring et de ses principaux composants (i.e. monitoring, analyse et réparation). Ils ont tout d'abord montré, l'efficacité d'AOP4CSM (overhead très réduit). Ensuite, ces expérimentations ont montré les mérites de notre approche d'analyse en termes de précision et de rappel, en comparaison avec les approches basées sur les seuils. Ces résultats obtenus ont montré aussi l'exactitude de notre approche dans l'identification de la cause de la dégradation. En plus, nos expérimentations ont prouvé que D-CEP4CMA est efficace en termes de performance, de précision et de rappel, en comparaison avec les architectures centralisées et distribuées. Aussi, les résultats obtenus ont illustré que le temps de réparation est relativement réduit et ont montré que l'overhead CPU de CEP4Cloud est négligeable. Ces résultats ont démontré les avantages de CEP4Cloud en termes de temps de réparation et de nombre d'alarmes, en comparaison avec les approches de référence.

Erklärung

Ich versichere, dass ich meine Dissertation

Cross-Layer Cloud Performance Monitoring, Analysis and Recovery

selbständig, ohne unerlaubte Hilfe angefertigt und mich dabei keiner anderen als der von mir ausdrücklich bezeichneten Quellen und Hilfen bedient habe. Die Dissertation wurde in der jetzigen oder einer ähnlichen Form noch bei keiner anderen Hochschule eingereicht und hat noch keinen sonstigen Prüfungszwecken gedient.

Marburg, den _____

_____

Afef Mdhaffar

Dedication

This dissertation is dedicated to the memory of my dear mother who left us suddenly and accidentally during the final stages of this work. After your departure, I have just lost my love of life. I still see your smile, still hear your voice and still keep trying to achieve my (our) dreams, as you always taught me. I am very close to realize our first dream. I hope you can see/feel that.

Acknowledgments

This thesis was partly supported by the German Ministry of Education and Research (BMBF) and the German Academic Exchange Service (DAAD). It was elaborated under a jointly supervised thesis agreement between the University of Marburg, Germany and the National School of Engineers of Sfax (ENIS), University of Sfax, Tunisia. This gave me the great chance to have two excellent supervisors: Prof. Dr. Bernd Freisleben (University of Marburg) and Prof. Dr. Mohamed Jmaiel (ENIS).

Mohamed Jmaiel has supervised my Master thesis and showed me the basis of research. He introduced me to Bernd Freisleben, in the context of a DAAD workshop. This gave me the opportunity to visit a German research group.

I feel very lucky that Bernd Freisleben supervised my research. This PhD thesis is largely due to his outstanding support; Bernd taught me a lot about research and showed me the art of scientific writing. This dissertation would look completely different without him.

Riadh Ben Halima deserves special thanks, as he shared with me his knowledge and his research experience.

I am grateful to the committee members for accepting to evaluate this PhD thesis.

I also want to thank Thomas Gebhardt and Markus Böttner from the local computer center at the University of Marburg, for their valuable technical support and their patience, when a new machine fails again.

Additionally, I want to thank Cornelia Hanzlik-Rudolph and Anke Bahrani from the DAAD office (north Africa section), for their availability.

Special thanks go to Naïma Kolsi-Benzina, Marc Strickert and Amira Abdel-Aziz for proofreading parts of this dissertation.

I also want to thank my colleagues at the Distributed Systems Group (University of Marburg) and at the ReDCAD laboratory (University of Sfax) for our interesting discussions.

Mechthild Keßler, our secretary at the University of Marburg, deserves particular thanks, for efficiently dealing with all administrative tasks.

Most of all, I want to thank my father Anouar for his extraordinary support during the last 374 months. I owe to him what I am today.

Distinctive thanks go to my sister Rim, my brothers Habib and Mohamed Ali, and my aunts Naïma and Sondos, for their care and valuable support.

Last but certainly not least, I thank my nephews Mohamed, Zeineb and Karim for the innocent and sweet Skype discussions that made my life happier.

Contents

"The greatest challenge to any thinker is stating the problem in a way that will allow a solution."

Bertrand Russell

1 Introduction

1.1 Motivation

In the context of MIT's centennial (1961), the computer scientist John McCarthy predicted that computing may be offered as a public utility like water and electricity [33]. Several decades later, a new computing technology, called Cloud computing, has emerged to make McCarthy's prediction true. Indeed, Cloud computing has been designed to offer everything as a service. Its basic idea is very exciting since it makes accessing many thousands of computers possible and as easy as online shopping.

Contrary to traditional computing paradigms, Cloud computing offers many levels of services to customers. The most known ones are SaaS (Software as a Service), PaaS (Platform as a Service) and IaaS (Infrastructure as a Service). These services are delivered on demand, while following the "Pay as You Go" model (i.e., paying exactly what you use). This allows customers (i.e., Cloud users) to save a potentially large amount of money by renting the required infrastructure.

Today's Cloud computing environments are typically based on a layered architecture consisting of infrastructure, platform and software layers. Clouds consist of many hardware and software resources and are accessed by many concurrent users. In such a complex environment, performance-related problems are quite likely and rather the norm than the exception. Consequently, it is necessary to monitor and analyze performance parameters of Cloud computing environments to detect and rectify related problems.

This thesis focuses on defining, implementing and validating a reactive performance monitoring approach for Cloud computing environments. The target

solution should be able to efficiently detect and rectify performance-related problems, while monitoring and analyzing performance parameters.

1.2 Problem Statement

Designing a reactive monitoring solution for Cloud computing environments is a challenging task since it leads to three main research topics. The first topic deals with Cloud monitoring, the second topic is related to Cloud analysis, and the third topic deals with Cloud recovery.

Most existing monitoring approaches for Cloud services (SaaS layer) require access to the source code of the services being monitored and are typically operated by the provider. The first challenge to be solved is to define a non-invasive monitoring approach for Cloud services.

Related monitoring and analysis approaches have been designed to separately work for only one of the Cloud layers (infrastructure, platform, software) and thus do not consider the interactions between these layers. Exploiting the relationships between metrics across Cloud layers is promising in terms of accuracy, but expensive in terms of rapidity. Actually, a multi-layer monitoring and analysis approach generates a huge volume of data. Thus, it could be quite slow and could consume a lot of storage space. The second challenge of this thesis is to define a fast and an accurate analysis approach that exploits the interactions between Cloud layers.

Existing analysis approaches are based either on a centralized architecture or on a distributed architecture. Each architecture has some disadvantages and does not fit to an elastic Cloud. A centralized analysis component could easily become a bottleneck if the amount of monitored data exceeds its processing capacities (i.e., the size of the Cloud increases). On the other hand, distributed analysis architectures suffer from the potentially large number of messages exchanged between the distributed analysis components. They become unnecessary if the size of the Cloud goes down. The third challenging task to be solved is the definition of a dynamic analysis architecture that fits to the elasticity property of Clouds (i.e., scale up/down).

Related recovery approaches are usually based on assigning a single recovery action to a given performance-related problem and they do not check the success of the applied action. This could lead to bad results if the applied recovery action has failed. The fourth challenge of this thesis is to define a recovery approach that is able to identify all possible recovery actions to a given performance-related problem and validate the success of the applied recovery action.

The last challenging task to be solved is to combine the monitoring, analysis and recovery approaches to build an efficient cross-layer reactive monitoring solution for Cloud computing environments.

1.3 Contributions

The contributions of this thesis are summarized below:

- A novel monitoring approach for Cloud services is defined. The proposed approach is called AOP4CSM for "**A**spect-**O**riented **P**rogramming for **C**loud **S**ervice **M**onitoring". AOP4CSM uses Aspect-Oriented Programming (AOP) as a method to collect, in a non-invasive manner, quality of service (QoS) parameters such as the execution and the response times of Cloud services at the SaaS layer. AOP4CSM does not require access to the source code of a service, and can be installed by the client.

- A novel cross-layer monitoring and analysis approach for Cloud computing environments is proposed. The defined approach deals with performance-related problems. It is called CEP4CMA for "**C**omplex **E**vent **P**rocessing for **C**loud **M**onitoring and **A**nalysis". CEP4CMA offers accurate diagnosis and does not require any storage space for recording monitored events. It is based on the methodology of Complex Event Processing (CEP). The novelty of CEP4CMA is that the CEP queries are derived from a comprehensive analysis of the relationships between monitored metrics across Cloud layers. The correlations between the monitored metrics on different Cloud layers are obtained via a set of experiments and well known statistical methods. The results of our correlation study allow us to reduce the number of monitored parameters. Furthermore, they are used to perform a Root Cause Analysis (RCA) to identify the causes of performance-related problems.

- A novel dynamic architecture for Cloud performance monitoring and analysis based on CEP is proposed. The defined architecture is called D-CEP4CMA for "**D**ynamic **C**omplex **E**vent **P**rocessing for **C**loud **M**onitoring and **A**nalysis". The basic idea of D-CEP4CMA is to dynamically switch between different CEP architectures depending on the current conditions of the observed Cloud environment. It is based on two novel algorithms to decide whether to activate particular architectural components. D-CEP4CMA's algorithms are deduced from an experimental study of three different CEP architectures for Cloud monitoring and analysis, a centralized one and two distributed ones.

- A novel multi-level recovery approach is proposed. It is based on a new action manager framework that (1) assigns a set of repair actions to each performance-related problem and (2) dynamically selects and applies the most adequate one. The success of the applied action is checked in our proposal.

- A novel cross-layer reactive performance monitoring approach for Cloud computing environments is presented.

1.4 Publications

The following papers have been published in the context of this thesis:

Journal Publications

- Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. D-CEP4CMA: A Dynamic Architecture for Cloud Performance Monitoring and Analysis via Complex Event Processing. *International Journal of Big Data Intelligence*, 1(1/2):89–102, 2014

- Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. Reactive Performance Monitoring of Cloud Computing Environments. 2014. *Submitted for publication*

Conference Publications

- Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. CEP4Cloud: Complex Event Processing for Self-Healing Clouds. In *Proceedings of the 23rd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 62–67, Parma, Italy, 2014. IEEE Press

- Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. CEP4CMA: Multi-Layer Cloud Performance Monitoring and Analysis via Complex Event Processing. In *Proceedings of the 2nd International Conference on Networked Systems*, volume 8593 of *Lecture Notes in Computer Science*, pages 138–152, Marrakech, Morocco, 2014. Springer

- Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. A Dynamic Complex Event Processing Architecture for Cloud Monitoring and Analysis. In *Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science*, pages 270–275, Bristol, UK, 2013. IEEE Press

- Afef Mdhaffar, Riadh Ben Halima, Ernst Juhnke, Mohamed Jmaiel, and Bernd Freisleben. AOP4CSM: An Aspect-Oriented Programming Approach for Cloud Service Monitoring. In *Proceedings of the 11th IEEE International Conference on Computer and Information Technology*, pages 363–370, Paphos, Cyprus, 2011. IEEE Press

- Meriam Mahjoub, Afef Mdhaffar, Riadh Ben Halima, and Mohamed Jmaiel. A Comparative Study of the Current Cloud Computing Technologies and Offers. In *Proceedings of the 1st International Symposium on Network Cloud Computing and Applications*, pages 131–134, Toulouse, France, 2011. IEEE Press

- Afef Mdhaffar, Soumaya Marzouk, Riadh Ben Halima, and Mohamed Jmaiel. A Runtime Performance Analysis for Web Service-Based Applications. In *Proceedings of the 1st Workshop on Engineering SOA and the Web held in conjunction with the 10th International Conference on Web Engineering*, volume 6385 of *Lecture Notes in Computer Science*, pages 313–324, Vienna, Austria, 2010. Springer

The paper entitled **"CEP4Cloud: Complex Event Processing for Self-Healing Clouds"** received a **"Best Paper Award"** at the "23rd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises" (WETICE 2014), Parma, Italy, in 2014.

1.5 Structure of the Thesis

The remainder of this thesis is organized as follows.

Chapter 2 introduces basic concepts related to this thesis. This includes autonomic computing, Cloud computing and virtualization. Furthermore, Chapter 2 presents approaches and techniques used during the course of this work, such as aspect-oriented programming, complex event processing and root cause analysis.

Chapter 3 describes and discusses related work. It focuses on monitoring, analysis and recovery approaches for Cloud computing environments. Based on the drawn conclusions, Chapter 3 identifies the requirements of this thesis.

Chapter 4 details the design of the cross-layer reactive monitoring approach for Cloud computing environments, called CEP4Cloud. First, it describes the architecture of CEP4Cloud. Second, it details the main components of CEP4Cloud: the multi-layer monitoring agent, the CEP-based cross-layer analysis agent and the action manager framework.

Chapter 5 details the implementation of CEP4Cloud and its main components. First, it presents a high-level view of CEP4Cloud's structure. Then, it gives implementation details regarding the monitoring, analysis and recovery approaches

Chapter 6 presents and discusses experimental results. It illustrates the merits of CEP4Cloud compared to baseline approaches.

Chapter 7 concludes this thesis and outlines areas of future research.

"The greatest obstacle to discovery is not ignorance – it is the illusion of knowledge."

Daniel J. Boorstin

2 Fundamentals

2.1 Introduction

This chapter introduces the basic concepts related to this thesis. It covers autonomic and Cloud computing, virtualization, and the methodologies / techniques used in the context of this research work, such as Aspect-Oriented Programming (AOP), Complex Event Processing (CEP) and Root Cause Analysis (RCA). Section 2.2 deals with autonomic computing, while Section 2.3 focuses on Clouds. An overview of virtualization is given in Section 2.4, since it plays a principal role to build a Cloud computing environment. Afterwards, the methodologies used to define and develop the proposed approaches are presented. An introduction to Aspect-Oriented Programming is given in Section 2.5. The techniques of Complex Event Processing and Root Cause Analysis are described in Sections 2.6 and 2.7, respectively. Section 2.8 summarizes this chapter.

2.2 Autonomic Computing

The idea of autonomic computing has firstly been introduced by Paul Horn, IBM's senior vice president of research [52]. Horn has used a biological connotation to describe the autonomic computing paradigm. He viewed an autonomic system as the human nervous system that it is able to self-govern the heart rate and the body temperature. Indeed, IBM has defined autonomic computing as a computing environment which is able to (1) "*know it self*", (2) "*re-configure itself under changing conditions*", (3) "*heal it self, when a problem occurs*", (4)"*optimize it self*" and (5) "*protect it self from dangerous situations*" [35]. So, autonomic

computing systems should be able to achieve the four following objectives [44, 77]:

Self-configuration: An autonomic system is able to configure and re-configure it self according to changing and unpredictable conditions. It automatically adapts to the needs of the environment (e.g., platform, user).

Self-healing: An autonomic system is able to detect, diagnose and repair potential issues, without stopping to function.

Self-optimization: An autonomic system is able to monitor and automatically optimize / tune its resources to meet the defined requirements (e.g., improve quality of service, improve performance).

Self-protection: An autonomic system should be able to prevent security issues from occurring. It has to protect it self from malicious attacks.

As shown in Figure 2.1, these self-X objectives are related to the properties of software agents, identified by Wooldrige and Jennings [106]: Autonomy, Social ability, Reactivity and Pro-activeness [44].

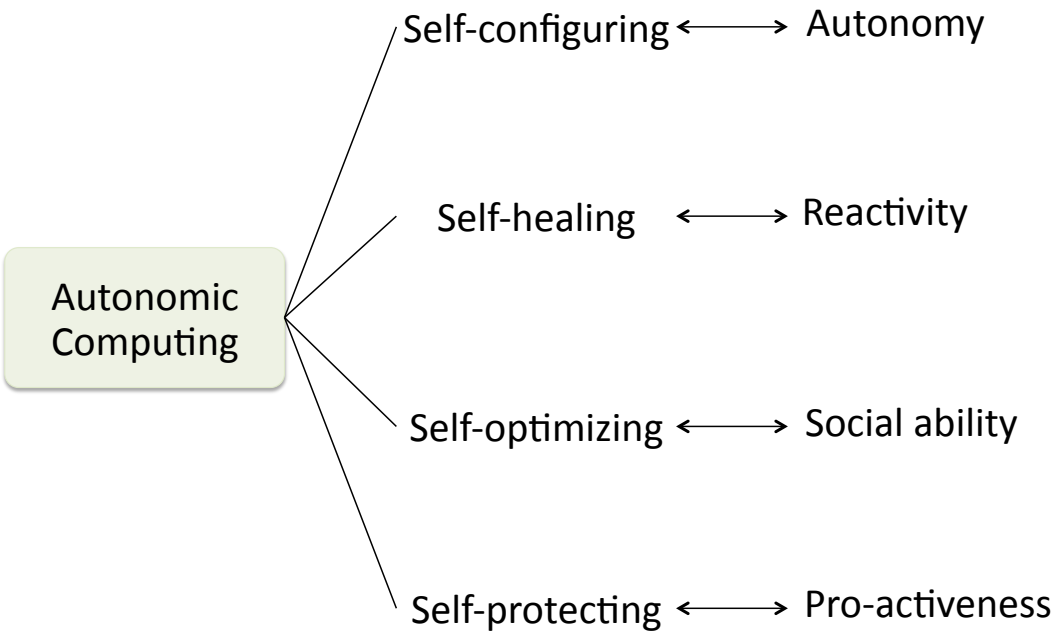

Figure 2.1: Autonomic computing

Autonomy is the capability of operating without direct human intervention or any other kind of external control [44].

Social ability characterizes agents which interact with other agents or humans via some communication protocols [44].

Reactivity is the capability of an agent to evaluate itself and react in a timely fashion to changes [44].

Pro-activeness characterizes agents that are able to predict problems and prevent them from occurring [44].

To achieve the Self-X objectives described above, an autonomic system is typically based on the MAPE-K (**M**onitoring, **A**nalysis, **P**lanning, **E**xecution - **K**nowledge) loop. It firstly starts by collecting events. Afterwards, it analyzes the gathered data to describe its state and plans recovery actions, if a failure has been detected. The planned recovery action will then be executed. These phases are usually based on some knowledge. The MAPE-K loop is detailed in the next section.

MAPE-K Loop

Figure 2.2 depicts the MAPE-K loop. It consists of four main steps and makes use of some knowledge about the managed element.

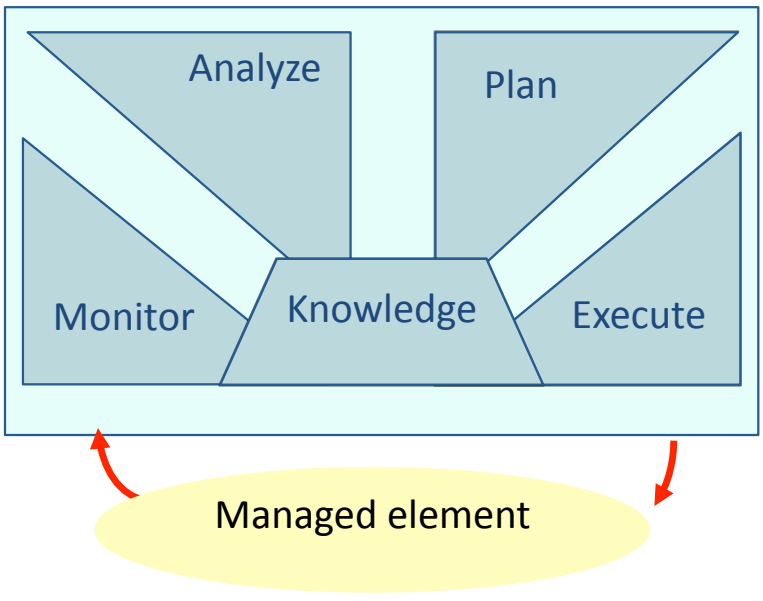

Figure 2.2: MAPE-K loop [52]

The main elements of the MAPE-K loop are described below.

Monitor: The first step of the MAPE-K loop allows us to monitor the managed element, while collecting related metrics.

Analyze: The second step of the MAPE-K loop allows us to analyze and diagnose the monitored metrics, collected during the first phase. The analysis phase generates diagnosis reports describing the state of the managed system. Moreover, it detects failures, triggers alarms and notifies the planning phase (i.e., third step of the MAPE-K loop) to ask for recovery actions.

Plan: The third step of the MAPE-K loop is the planning phase. It allows us to identify the best recovery action to repair the detected failure.

Execute: The last step of the MAPE-K loop is in charge of executing the planned recovery actions.

Knowledge: The knowledge element of the MAPE-K loop is used to describe the characteristics of the managed system, such as the system configuration [52].

2.3 Cloud Computing

The basic idea of Cloud computing is not new. It refers to a long-held dream of computing as a utility [3, 33], that has finally been established [4]. Indeed, Cloud computing is a new computing model that provides hardware and software resources as utilities [110].

The term Cloud does not have a standard definition. Therefore, many researchers have tried to standardize the definition of Cloud computing. For instance, Armbrust et al. [4] discussed the most known Cloud definitions. In this thesis, we adopt the definition provided by the National Institute of Standards and Technology (NIST), since it covers the main characteristics of Cloud computing.

Cloud Computing Definition (provided by NIST)

NIST defines Cloud computing as *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [71].

The remainder of this section describes the main aspects of Cloud computing. First, we detail the five principal characteristics of Clouds. Second, we describe the layered architecture of Cloud computing. Third, we present the most known

Cloud services. Finally, we give an overview about the deployment models of Cloud computing.

2.3.1 Essential Characteristics of Cloud Computing

On-demand self-service: Cloud services are available on-demand and could be rented by Cloud customers at any time, without human intervention [71].

Broad network access: Cloud services are available via network connections and accessed from all kinds of devices: thick and thin client platforms (e.g. workstations, laptops, tablets and mobile phones) [71].

Resource pooling: The model of Cloud computing is multi-tenant. Cloud resources are pooled to serve many Cloud users. They are allocated and reallocated, according to the demands of Cloud consumers [71].

Rapid elasticity: Cloud resources are elastically allocated and released to meet the requirements of Cloud users (e.g. scale up when the resources are over-used) and save resources (e.g., scale down when the resources are not used), respectively [71].

Measured service The usage of Cloud resources is monitored and reported. The generated reports can be accessed by the customers and the provider of the Cloud. Based on a pay-per-use strategy, Cloud resources are automatically optimized [71].

2.3.2 Cloud Computing Architecture

Figure 2.3 depicts the layered architecture of Cloud computing.

It consists of four main layers: (1) the physical infrastructure layer, (2) the virtualization layer, (3) the platform layer and (4) the software layer. Cloud layers are detailed below.

The physical infrastructure layer is composed of hardware resources such as physical machines, routers, switches and power. Physical resources are usually organized in racks and interconnected via switches or routers.

The virtualization layer is in charge of managing virtual machines (VMs). It is based on a virtualization technology such as Xen, KVM and VMware, to create and manage virtual resources.

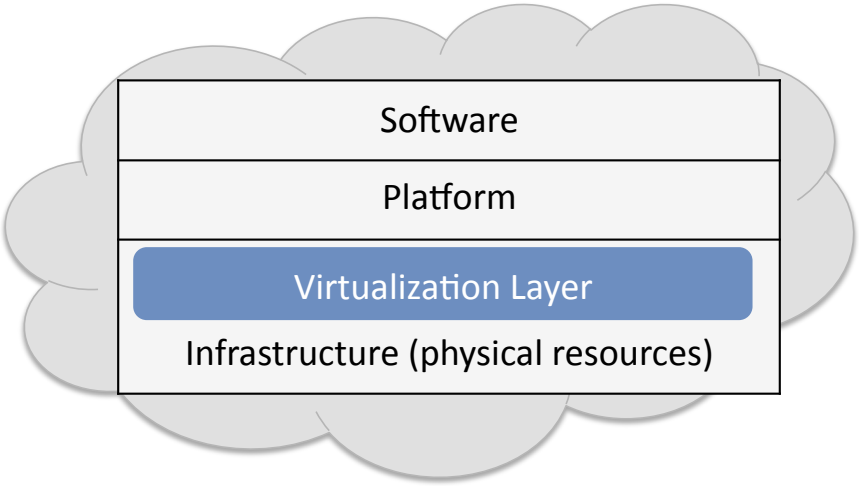

Figure 2.3: The layered architecture of Cloud computing

The platform layer is running on top of the virtualization layer. It is composed of operating systems and running platforms such as database and web servers.

The software layer is the highest layer of the Cloud computing architecture. It consists of applications and services, such as web services.

2.3.3 Cloud Services

The main idea of Cloud computing is to offer everything as a service (XaaS). As shown in Figure 2.4, the most known Cloud services are related to the Cloud computing architecture. They are:

Infrastructure as a Service (IaaS) is a Cloud service that offers, in an on-demand fashion, a virtualized infrastructure (i.e., virtual machines) as a service, through the use of virtualization technologies. The most known IaaS are Amazon EC2 [59] , Flexiscale [32] and GoGrid [97] .

Platform as a Service (PaaS) is the second most known Cloud service. It provides a software platform as a service, such as Google's App Engine [20] and Microsoft Azure [103]. Google's App Engine allows us to develop and run Web application on Google's platform.

Software as a Service (SaaS) is a Cloud service that offers a software (i.e. application) as a service, such as e-mail clients (e.g., Gmail) and document management software (e.g., Google Docs).

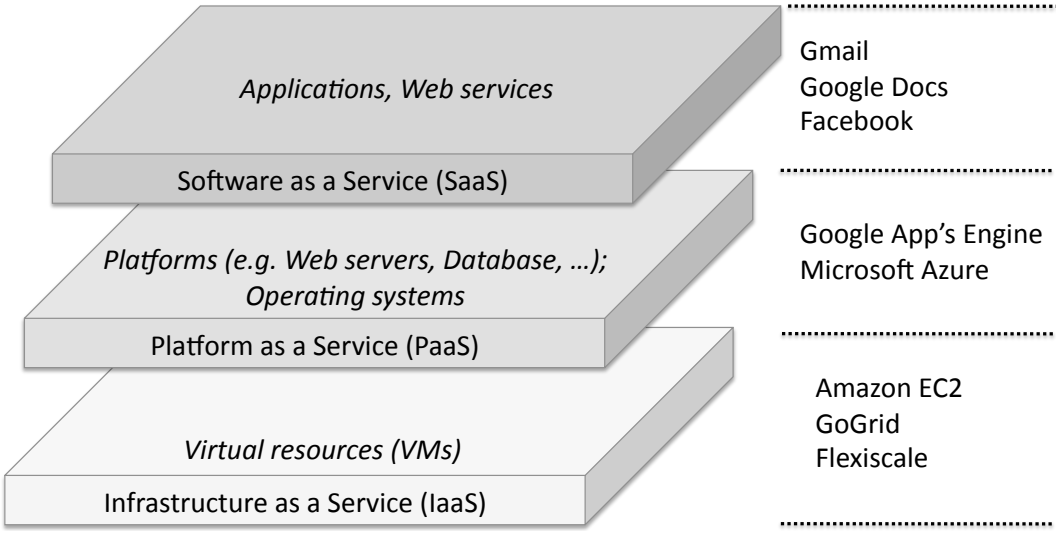

Figure 2.4: The most known Cloud services

Since Cloud computing is based on the "Pay as You Go" model, it allows Cloud customers to save a potentially large amount of money, by renting the required software / hardware resources.

2.3.4 Deployment Models of Cloud Computing

As shown in Figure 2.5, there are four deployment models of Cloud computing: private Cloud, community Cloud, public Cloud and hybrid Cloud. They are described below.

Private Cloud

The Cloud environment is exclusively used by a single (private) organization composed of many consumers. It is usually owned and managed by the same organization and/or another Cloud provider [71].

Community Cloud

A community Cloud environment is exclusively used by a specific community of consumers belonging to several organizations, that share common concerns such as mission and policy. The community Cloud environment could be owned and managed by one or many organizations of the community, or another Cloud provider (i.e., a third party), or a combination of them [71].

Public Cloud

A public Cloud environment can be used by everyone. It is owned and managed by one or many organizations [71].

Hybrid Cloud

A hybrid Cloud consists of many (two or more) Cloud environments (private, community, or public) that share standardized technology to enable the portability of applications and data [71].

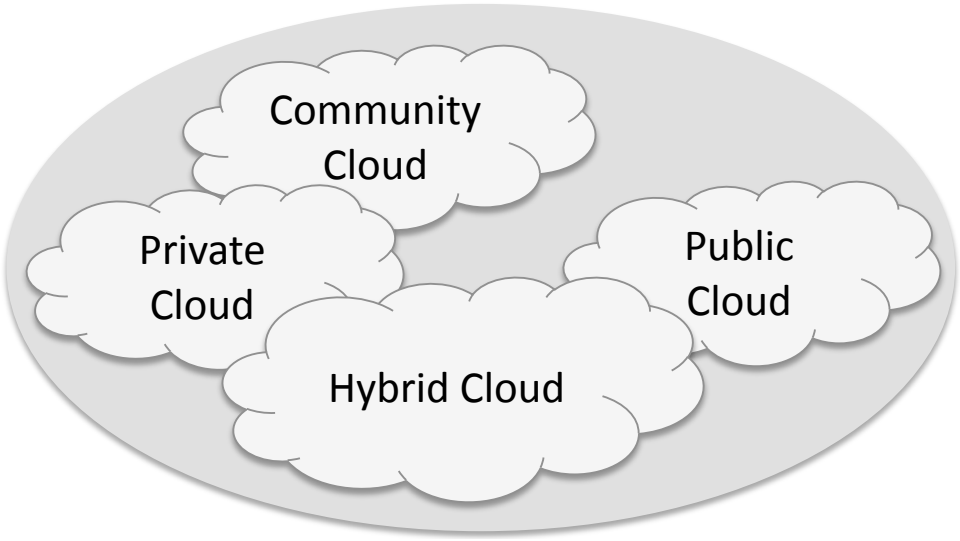

Figure 2.5: Deployment models of Cloud computing

Existing Cloud computing platforms are clustered into two groups: commercial and open source platforms. For instance, Amazon EC2 is a commercial IaaS, while OpenStack is an open source IaaS. For research and developing purposes, it is more convenient to use an open source Cloud since it allows developers to access the source code. The most prominent examples of open source Cloud computing platforms are OpenStack, Eucalyptus, OpenNebula and CloudStack. Several research work have compared and discussed the merits and issues of these open source Clouds. For instance, Sempolinski et al. [90] provides a comparison between Eucalyptus, OpenNebula and Nimbus.

In this research work, we use OpenStack, since it meets our needs. The next section details OpenStack.

2.3.5 OpenStack

OpenStack [99] is an open source IaaS Cloud, licensed under the Apache 2.0 License. It has been founded by "Rackspace Hosting" and "NASA". The Open-

Stack project is now managed by the OpenStack Foundation, established in September 2012 [51]. OpenStack has released ten OpenStack releases: from Austin (October 2010) to Juno (October 2014). The eleventh OpenStack release is called Kilo. Kilo is still under development and will be released in April 2015[1]. OpenStack is written in Python and can be used to build IaaS Clouds.

Openstack is based on a modular architecture. It consists of seven main services, which can be clustered into five groups. As shown in Figure 2.6, the first group includes compute services, while the second one is composed of networking services.

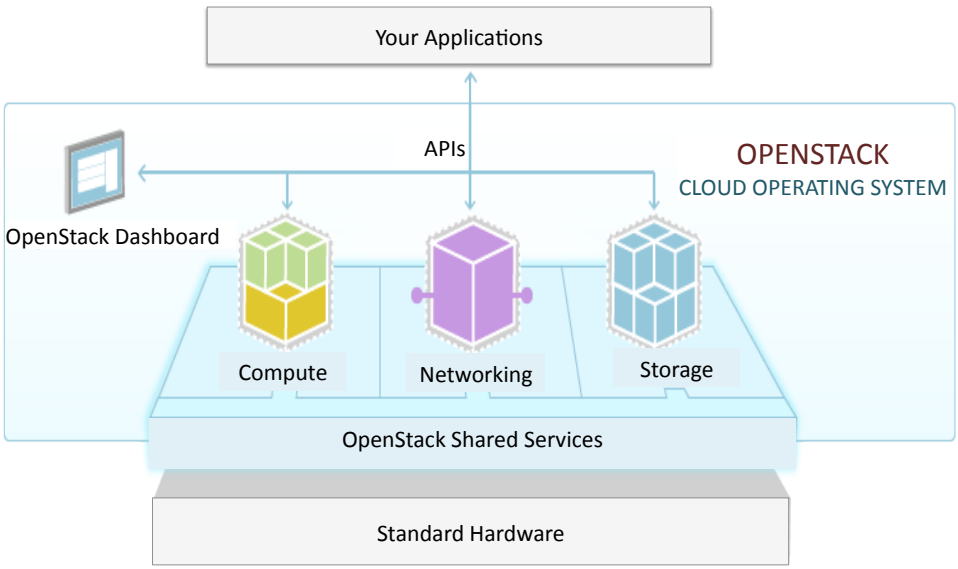

Figure 2.6: OpenStack architecture [2]

The third group includes the storage services, while the fourth one consists of the OpenStack dashboard. The last one is composed of shared services. These groups and their components are summarized in Table 2.1 and detailed below.

Compute service

Compute services are used to manage instances (i.e., virtual machines), while retrieving images and associated metadata. They are also called "Nova" services.

Networking services

The networking services are in charge of managing the virtual network of the created instance (e.g., assigning IP addresses to instances).

[1] https://wiki.openstack.org/wiki/Releases
[2] http://www.openstack.org/software/
[3] OpenStack includes also telemetry, orchestration and database services

Table 2.1: OpenStack services

Service Group	Service Name	Service Identifier
Compute	Compute	Nova
Networking	Network	Quantum / Neutron
Storage	Block Storage	Cinder
	Object Storage	Swift
Dashboard	Dashboard	Horizon
Shared services[3]	Identity	Keystone
	Image	Glance

Storage services

The storage services include the object storage and block storage services. The object storage service, also known as "Swift", allows users to store and/or retrieve data. The block storage service, called "Cinder" is responsible of providing storage volumes for the compute services.

Dashboard

The dashboard is the web management interface of OpenStack. It is called "Horizon" and can be used to access OpenStack services, such as launching virtual machines and creating snapshots. Figure 2.7 shows a screen snapshot of "Horizon".

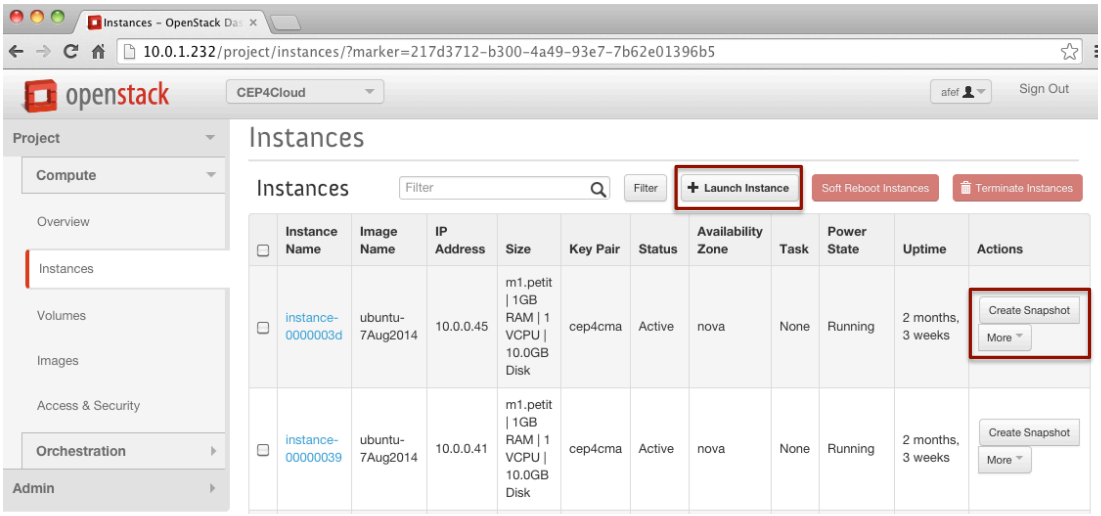

Figure 2.7: OpenStack dashboard: Horizon

Shared services

OpenStack makes use of many shared services, which allow the communication between OpenStack components. The most important shared services of OpenStack are the image and the identity services. The image service, also called "Glance", is used to store the images. The identity service, known as "Keystone", is the authentication system of OpenStack. It is used to manage users, tenants and permissions.

OpenStack services cooperate with each other to achieve the operations required by the users. Figure 2.8 depicts the relationships between OpenStack services. It shows the conceptual diagram of OpenStack.

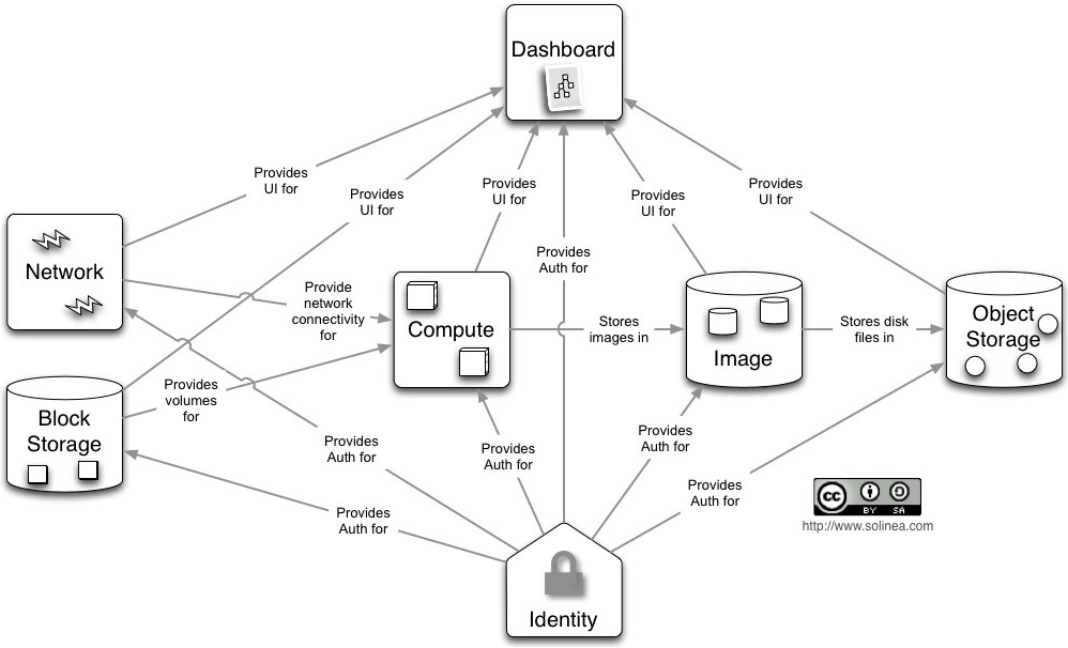

Figure 2.8: The conceptual diagram of OpenStack [4]

As already mentioned, Cloud computing environments are typically based on the use of vitualization technologies, to create and manage virtual resources. Virtualization technologies are detailed in Section 2.4.

2.4 Virtualization

Virtualization is a technique that allows us to logically separate between the physical resources and the running software. It provides an abstraction layer between running applications and the hardware. Indeed, the virtualization tech-

[4] http://docs.openstack.org/training-guides/content/developer-getting-started.html

nology makes the software independent of the hardware [42]. In particular, virtualization can be used to run many operating systems on the same physical machine. The running operating systems correspond to virtual machines. This corresponds to a specific type of virtualization, called "Server and Machine Virtualization". There are many other types of virtualization like application, desktop and network virtualization. The next section gives and describes existing types of virtualization.

2.4.1 Types of Virtualization

There are six principal types of virtualization. They are described below [42]:

Application virtualization

Application virtualization allows us to compile applications into a byte code that is independent of the machine. The produced byte code can be used on any system that has an execution environment, based on the same virtual machine. The most known example of this type of virtualization is the byte code generated by the compilers for the Java programming language [42].

Desktop virtualization

Desktop virtualization is a technique that allows us to display a graphical desktop of one machine on another machine. The most known example of desktop virtualization is VNC (Virtual Network Computing). Von Hagen [42] believes that the term "desktop virtualization" does not really describe a virtualization technique. Actually, displaying the graphical desktop of a remote machine on another machine is not related to virtualization, since the applications are still running on the same and single (remote) machine [42].

Network virtualization

Network virtualization is a logical abstraction of physical network resources. It allows us to logically refer to network resources [42]. Chowdhury et al. [18] provide a detailed description of a network virtualization environment (NVE) and compare a NVE to a traditional network model. The classic example of network virtualization is VPN (Virtual Private Network). VPN allows users to logically connect multiple sites.

Storage virtualization

Storage virtualization is well known and has been defined since several years. It represents the logical abstraction of physical storage. LVM (Logical Volume Manager) is the most familiar example of storage virtualization [42].

System-level virtualization

The system-level virtualization, also called operating system virtualization, is based on the concept of change root (chroot). As shown in Figure 2.9, system-level virtualization allows users to run many virtual servers on a shared (single) copy of an operation system kernel [42].

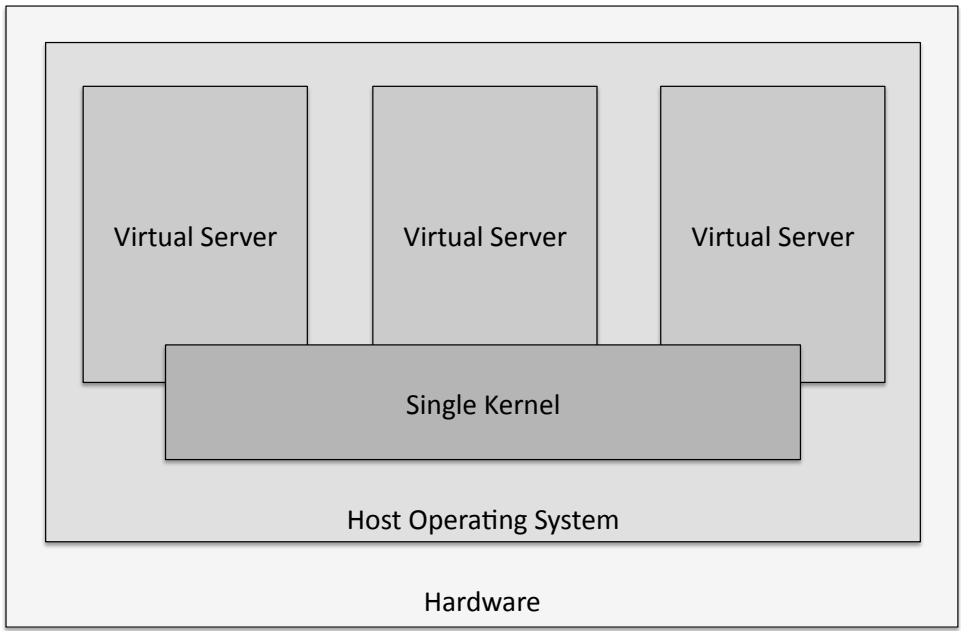

Figure 2.9: System-level or operating system vitualization [42]

Server virtualization

Server virtualization, also called machine virtualization, is the most famous virtualization type [42]. It allows users to run many virtual machines on another operating system. The running virtual machines have their own operating system. They are able to execute their applications within their own operating systems. Hardware resources are logically assigned to virtual machines. In contrast to system-level virtualization, server virtualization does not require to share the same kernel. Therefore, it is widely used. There are several approaches to server virtualization. The most known ones are detailed below:

- **Full virtualization:** It consists of abstracting all physical hardware and running unmodified operating systems [17], such as Virtualbox.
- **Hardware-assisted virtualization:** It allows us to run an unmodified operating system [17]. An example of a Hardware-assisted virtualization is KVM.

- **Para-virtualization:** It consists of running a modified operating system and does not require to virtualize all physical hardware, such as Xen [17].

Since Xen is used during the course of this thesis, it is detailed in the next section.

2.4.2 Xen

The Xen Virtual Machine Monitor (VMM) is a key element of the XenoServer project [34], launched at the University of Cambridge and led by Ian Pratt. Xen supports the capability of running multiple operating systems on a single physical machine [42]. It is based on para-virtualization [8].

Xen Architecture

Xen stands between the hardware and the operating system. A Xen system is composed of three core components [17]:

- The hypervisor
- The kernel
- The userspace applications

As shown in Figure 2.10, the core components of a system involving Xen are organized in a layered fashion.

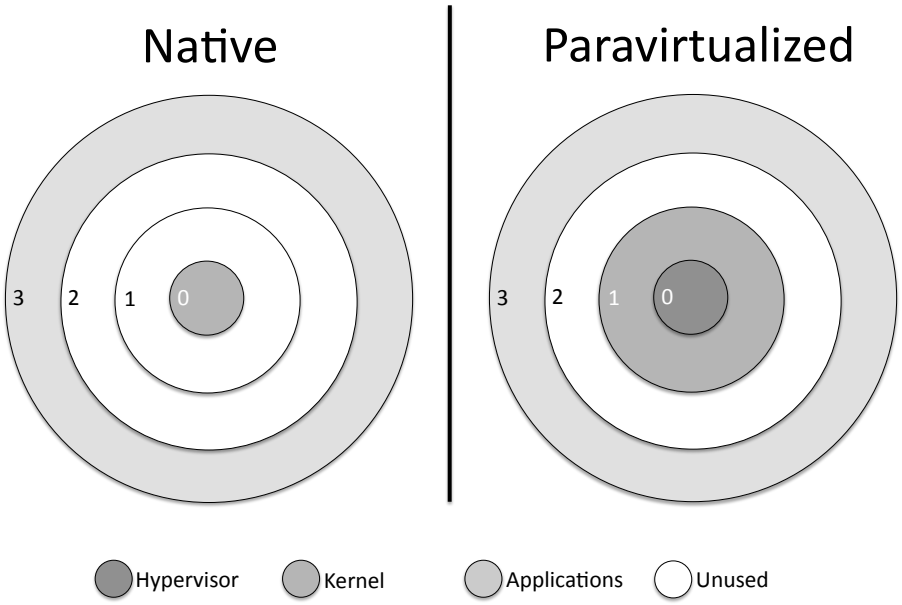

Figure 2.10: Ring usage in native and paravirtualized systems [17]

The operating system kernel is running in Ring 1 (i.e., under Xen), instead of ring 0 (i.e., Native) in the context of 32 bit hardware. Thus, the kernel is protected from applications and other kernels; and is able to access memory allocated to applications, which are running in Ring 3. The hypervisor is running in Ring 0. It is protected from kernels (in Ring 1) and applications in Ring 3 [17].

The hypervisor is used to start guests, which are running in environments, called domains. There are typically two kinds of domains: Domain 0 (Dom0) and Domain U (DomU). Dom0 has high privileges and boots after starting Xen. DomU is an unprivileged (U) domain and is started via Dom0 [17].

To manage Xen systems, we use a toolstack such as Xl [107], Libvirt [58] and Xapi [80]. Xl is the default toolstack of Xen. It is compatible with the **xm** CLI. Libvirt is a virtualization toolkit, which is used to manage many virtualization technologies. It is able to communicate with xm. Xapi is usually used for Clouds. It is compatible with the **xe** CLI. We used Xapi to build our OpenStack Cloud environment.

During the course of this thesis, we used three principal techniques:

- Aspect-Oriented Programming (AOP)
- Complex Event Processing (CEP)
- Root Cause Analysis (RCA)

The used techniques are detailed in Sections 2.5, 2.5 and 2.7.

2.5 Aspect-Oriented Programming

In 1997, Kiczales et al. defined and developed a new programming paradigm, called Aspect-Oriented Programming (AOP) [54]. AOP complements previous programming paradigms like Object-Oriented Programming (OOP), while introducing the concept of crosscutting concerns [81].

According to [53], Aspect-Oriented Programming (AOP) has the same significance for crosscutting concerns as Object-Oriented Programming (OOP) has for encapsulation and inheritance. The authors argue that crosscutting concerns can be programmed in a modular way and one can benefit from this modularity by "simpler code that is easier to develop and maintain, and that has greater potential for reuse" [53].

Indeed, AOP is based on a set of "aspects". An aspect is a unit of modularity, which implements a concern. We usually use an aspect-oriented programming language to define aspects and integrate them within the functional code. The integration process is called "weaving" and is performed via the use of an "Aspect Weaver". There are several implementations of AOP. The most known and documented one is AspectJ [81]. It is a free Java implementation of AOP. AspectJ extends Java to support the concept of aspect. It adds the new keyword "aspect"

[50]. The definition of an aspect is based on three keywords: joinpoint, pointcut and advice. They are detailed below:

- **A join point** is a well-defined point in the execution trace of a program. For instance, in AspectJ, a join point could be a call of a method or the execution of an exception handler [50].

- **A pointcut** is a query that selects a set of join points and collect their context, where the crosscutting function should be executed. For instance, a pointcut selects the call of a method M1 and captures its context. The context of M1 could be the return value or the type of M1's parameters [50].

- **The advice** is a piece of code that is executed when the defined join point is reached. It implements the crosscutting functionality. It is equivalent to a method in the context of object oriented programming. The main difference between a method (OOP) and an advice (AOP) is the fact that an aspect is never explicitly called [50]. In AspectJ, there are three advices: (1) before advice, (2) around advice and (3) after advice [81].

The *pointcut language* defines a set of join points where the advice are integrated into the code. Therefore, aspect-oriented programming eases the development of reusable and maintainable code [63].

Listing 2.1 shows an example of an aspect written in the AspectJ language. It allows us to send an SMS after every bank transaction. The pointcut "smsfuntion()" (see Line 3 of Listing 2.1) specifies where the "send sms" concern will be executed. This pointcut captures the call of any method belonging to the class "Account" (i.e. all bank transactions). The advice (see Lines 5, 6 and 7 of Listing 2.1) specifies when the concern will be executed (line 5) and what is the main behavior of the concern (line 6). Thus, this aspect allows us to send an SMS after every bank tansaction.

```
1 Public aspect SMSBank{

3 Pointcut smsfunction(): execution(* Account.*(float));

5 After : smsfunction(){
          sendSMS () ;
7     }
  }
```

Listing 2.1: Aspect: An example written in AspectJ

2.6 Complex Event Processing

Complex **E**vent **P**rocessing (CEP) is an approach to realize publish-subscribe systems. It can be used to monitor, process and analyze elementary events to deduce complex events describing the state of the monitored system.

The main functionality of a CEP system is to process events [39]. Processing events includes two main functions [39]:

- Consume events from the inputs of the system.
- Produce events on the outputs of the system.

An event describes the state of something (e.g., a parameter or a system) at a specific timestamp. There are two types of events: (1) elementary events and (2) complex events. Elementary events, also called simple events, are usually the inputs of a CEP engine. Complex events, also called composite events, represent the outputs of a CEP system. For instance, if the CEP engine receives the four following elementary events:

1. Church bells ringing,
2. A man wearing a nice suit,
3. A woman wearing a white dress,
4. People throwing rice.

Then, the CEP system generates a complex event, as output, indicating that there is a wedding [78].

2.6.1 CEP Architecture

Figure 2.11 [24] depicts a high-level view of a complex event processing system. As shown in Figure 2.11, a CEP system is based on three main actors:

- The event observers (also called sources)
- The CEP engine
- The event consumers (also called sinks)

The sources observe notifications of elementary events that have happened in the external world (i.e., the external monitored system). The CEP engine is in charge of processing (e.g., filtering, combining, joining ...) the received events to generate complex events describing the state of the monitored system. The generated complex events are consumed by the sinks (also called event consumers).

The CEP engine is usually based on a set of queries, to process elementary events.

The most prominent example of complex event processing systems is Esper [96]. It is an open source CEP system. Esper is presented in Section 2.6.2.

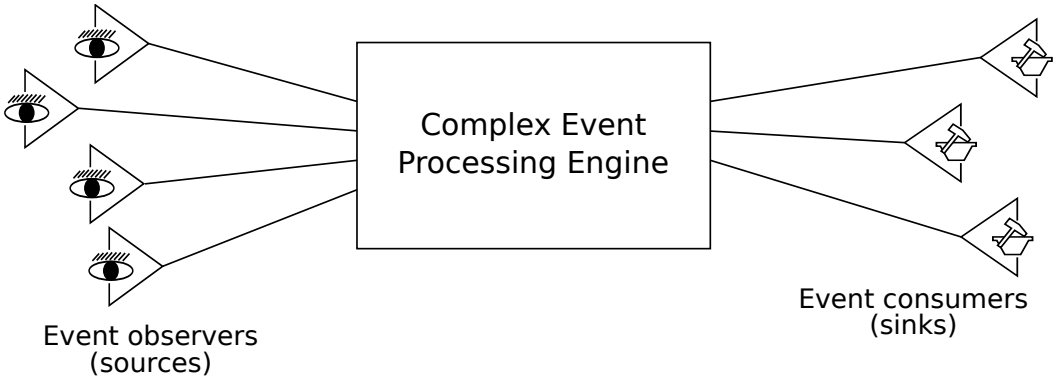

Figure 2.11: Complex event processing [24]

2.6.2 Esper

The Esper CEP engine is typically used to analyze and react to events. It has been applied in several domains such as business process management and automation, finance and network monitoring [96].

Esper implements EPL (Event Processing Language) queries. EPL is a SQL-like language. It runs queries on streams of events instead of tables. All SQL clauses (e.g., SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY) and SQL concepts of correlation (e.g., join, filtering) are available. Moreover, Esper supports pattern matching [96].

Esper is available in two versions:

- A Java version, called Esper (used during the course of this research work).
- A .Net version, called NEsper.

For more details about Esper, the reader is referred to the online tutorial of Esper [96].

The "Root Cause Analysis" approach has also been used during the course of this thesis. It is detailed in Section 2.7.

2.7 Root Cause Analysis

Root Cause Analysis (RCA) is an analytic approach that helps analysts to diagnose crisis situations, while identifying the cause of failures [12]. It is the process that allows us to discover what happened, why it happened and how to prevent / solve it [12].

The most prominent tools that can perform a root cause analysis are "5 Whys", "Pareto chart" and "Fishbone diagrams" [46].

In this thesis, fishbone diagrams are used to perform root cause analysis. Fishbone diagrams are detailed below.

Fishbone Diagram

The fishbone diagram, also called cause-effect diagram or Ishikawa diagram, was introduced by Ishikawa in 1960 [46]. The fishbone diagram is a visual method that follows the root cause analysis approach to detect failures and identify their causes. Cause-effect diagrams are more useful than "5 Whys", when the considered issue is very complex and involves a lot of data [92]. In a fishbone diagram, the causes are clustered into categories. Each group of causes is related to finer causes. Figure 2.12 shows an example of a fishbone diagram, created by the Xmind tool [108]. Xmind is Java tool that allows us to design several types of diagrams, such as fishbone diagrams.

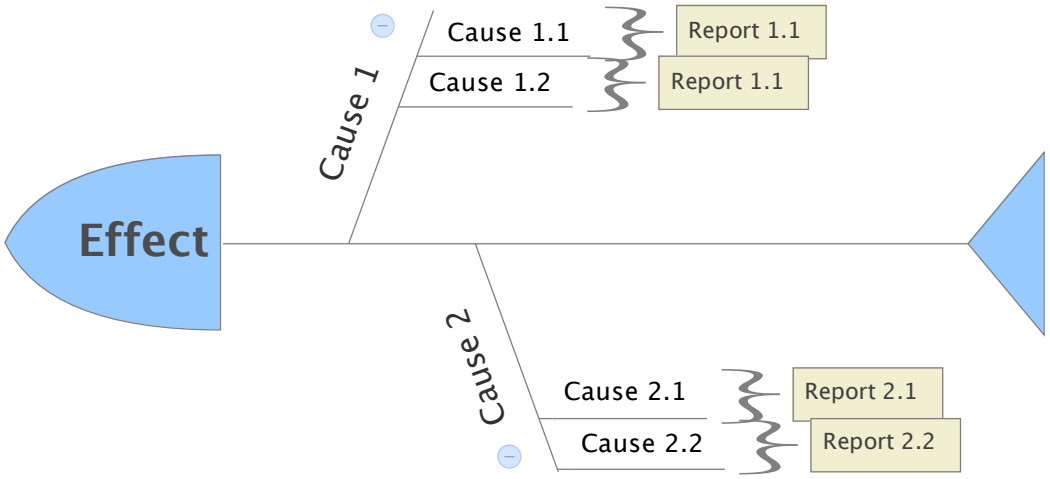

Figure 2.12: Fishbone diagram: An example

As shown in Figure 2.12, a fishbone diagram consists of three main elements:

- **The effect** represents the detected failure.
- **The cause** represents the origin of the detected failure (e.g. Cause 1) .
- **The report** describes the failure and its origins; and gives some hints to repair / prevent it (e.g. Report 1.1).

2.8 Summary

In this chapter, we have introduced basic concepts related to this thesis, such as autonomic and Cloud computing, and virtualization. Moreover, a description of approaches and techniques used in the course of this thesis has been given. This includes an overview about Aspect-Oriented Programming, Complex Event Processing and Root Cause analysis. The next chapter presents and discusses related work.

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."

Donald Knuth

3 Related Work

3.1 Introduction

In this chapter, related research projects with respect to self-healing approaches for Cloud computing environments are studied. These projects can be divided into three main groups. The first one includes solutions that focus on the first phase of the MAPE-K loop (i.e., monitoring). Related monitoring approaches are described and discussed in Section 3.2. The second group deals with analysis solutions that naturally include monitoring approaches. Existing monitoring and analysis solutions for Cloud computing environments in the presence of performance-related problems are presented and discussed in Section 3.3. The last group consists of self-healing approaches, addressing all the stages of the MAPE-K loop (i.e., monitoring, analysis and recovery). Section 3.4 presents and discusses related self-healing approaches. Based on the conducted study, the requirements of this thesis are defined. They are detailed in Section 3.5. Section 3.6 summarizes this chapter.

3.2 Monitoring

Existing monitoring approaches for Cloud computing environments can be grouped into four categories, according to the monitored Cloud layer. The first one includes monitoring approaches that collect QoS parameters at the software layer. The second category allows to monitor the platform layer, while the third one collects infrastructure-related parameters such as the CPU usage and the free memory. The fourth group includes multi-layer monitoring approaches that op-

erate at all Cloud layers and gather related parameters. Our related work study has shown that monitoring approaches usually use existing monitoring tools to gather parameters related to the infrastructure and platform layers. Also, we have learned from our study that QoS monitoring (software layer) is still a challenging topic. Moreover, our study has illustrated that the majority of existing monitoring approaches do not deal with all Cloud layers together. They usually consider them separately and do not propose a multi-layer monitoring approach for Cloud computing environments.

The rest of this section describes and discusses existing monitoring approaches for Cloud computing environments, while classifying them according to the monitored Cloud layer.

3.2.1 Monitoring: Software Layer

In this section, related work with respect to monitoring approaches for Web services and Cloud services is presented.

Web Service Monitoring

Thio et al. [102] propose a QoS monitoring framework for web service based applications. It clusters QoS parameters into two groups. The first group includes QoS metrics that can be measured from the Client side. The second group consists of QoS parameters that can be gathered from the server side. The authors define and discuss three different monitoring approaches to collect QoS parameters from both sides. The first proposed approach is called "low-level packet monitoring approach" and is based on monitoring tools such as libpcap [95] and winpcap [105]. The basic idea of this approach consists of capturing SOAP messages (both incoming and outgoing). This approach has some drawbacks, as the libpcap tool depends on the hardware. The second defined monitoring approach is "Proxy approach". It is based on the use of a proxy that is able to monitor activities exchanged between the Client and the service. To use this approach, we have to configure the client and server code. The third proposed monitoring approach is called "SOAP engine library modification approach". It extends the SOAP implementation API, both for the client and the server, to measure and log QoS parameter values. This enables the user to perform automated performance measurements. The approach depends on the used SOAP implementation: The required QoS monitoring extensions have to be deployed into the SOAP implementation used by the provider. Thus, this solution modifies the SOAP implementation.

Rosenberg et al. [85] propose a monitoring approach for Web services. Their approach makes use of monitoring tools such as Jpcap [48] for latency measurement. It relies on aspect-oriented and object-oriented programming techniques.

This proposal [85] does not require access to the Java source code of the service implementation, but it requires information related to the implementation of the monitored web service, such as endpoint and reference to WSDL.

Repp et al. [83] present an approach for monitoring performance across network layers such as HTTP, TCP, and IP. The basic idea of the proposed approach consists of measuring relevant points on different layers. These measurements points are used to deduce QoS parameters. This approach is aimed at detecting faults early. It reconfigures the system at real time while minimizing the substitution cost. It makes use of the windump tool that requires access to the hardware for monitoring. This means that its installation needs a special hardware configuration.

QOSH [9] is a self-healing middleware for web services. It allows to monitor QoS parameters of web services. The monitoring module of QOSH operates at the communication level. It is based on intercepting SOAP messages and adding QoS metadata to their headers. The QoS metadata depend on the collected QoS parameter. It could include relevant time stamps, if the monitored parameter is the response time. The monitoring module of QOSH consists of three components: the provider side monitor (PSM), the requester side monitor (RSM) and the logging manager. The PSM is in charge of intercepting and extending incoming messages, while the RSM intercepts and enriches outgoing SOAP messages. The logging manager is used to store data in the database. As this approach enriches SOAP messages with QoS information, QOSH modifies the client and the server implementation to allow QoS parameter evaluation.

Cloud Service (SaaS) Monitoring

Boniface et al. [13] propose a monitoring module that collects QoS parameters of SaaS. They use a monitoring application component (AC), which collects QoS parameters at both application and technical level. This AC has to be described and registered in the application repository in order to be used. Description and registration of AC makes this approach complicated and hard to install.

Cao et al. [15] propose a monitoring architecture for Cloud computing. It describes a QoS model that collects QoS parameter values such as response time, cost, availability, reliability and reputation. Their architecture has not been implemented yet. Technical details are not provided in the paper.

3.2.2 Monitoring: Platform Layer

Several monitoring tools have been developed to monitor the platform layer. All of them depend on the deployed platform. For instance, Jconsole [47] allows us to monitor JVM-Based platforms. Existing tools, dealing with platform monitoring, allow us to perfectly monitor the platform layer. They usually rely on engineering tasks and do not solve special research challenges. Therefore, it is judicious to choose one of these tools for platform monitoring. The choice is usually based on the nature of the Cloud platform layer.

3.2.3 Monitoring: Infrastructure Layer

About a decade ago, many approaches have been proposed to deal with hardware resources monitoring, such as Ganglia [36, 62], Nagios [74] and Chukwa [19, 82]. These approaches are now used for data center monitoring. They allow us to monitor the infrastructure layer of Cloud environments, while collecting all related parameters.

De Chave et al. [28] present a monitoring architecture for an IaaS Cloud. The authors intend to provide the first open source monitoring framework for Clouds. The proposed architecture, called PCMONS, makes use of open source monitoring tools such as Nagios. The challenge of this work is to deal with the majority of Cloud technologies. An integration layer that makes PCMONS easy to install in Cloud platforms such as OpenNebula and Eucalyptus is presented. PCMONS is limited to an IaaS Cloud; SaaS and PaaS are not considered.

DARGOS [79] is a monitoring system for multi-tenant IaaS Cloud computing environments. It is based on the publish-subscribe paradigm. The architecture of DARGOS consists of two main entities:

- Node Monitor Agent (NMA)
- Node Supervisor Agent (NSA)

The NMAs are in charge of collecting monitoring data from the local node and sending it to the interested node. The NSAs gather monitoring data from remote hosts and send it to the Cloud administrator via DARGOS API. The communication between NMAs and NSAs is performed via the use of a decentralized approach based on the Data Distribution System (DDS) data-centric middleware. The current implementation of DARGOS is based on the OpenStack IaaS Cloud (Nova project). The source code of OpenStack has been modified to support DARGOS. Thus, the current version of DARGOS is limited to the infrastructure layer and strongly depends on the architecture and implementation of the OpenStack Nova project.

3.2.4 Monitoring: All Cloud Layers

The majority of existing monitoring approaches deals with one Cloud layer. A few ones have been designed to operate at all Cloud layers, such as RMCM [91]. RMCM stands for "Runtime Model for Cloud Monitoring". It is a runtime monitoring model for Cloud computing environments. RMCM models all Cloud layers and collects related parameters. In the SaaS layer, RMCM monitors applications with respect to their design models and required constraints. For this issue, it converts the constraints to a corresponding instrumented code and deploys the resulting code at the appropriate location of the monitored applications. RMCM uses interceptors (as filters in Apache Tomcat and handlers in Axis) for service monitoring. This makes RMCM an invasive approach, since it modifies the source code of the applications. RMCM needs to be more flexible to let users decide about the monitoring metrics.

3.2.5 Discussion

Table 3.1 summarizes the main characteristics of studied monitoring approaches.

Table 3.1: Monitoring approaches for Cloud computing environments

Monitoring approaches	Cloud layers			
	Software		Platform	Infrastructure
	invasive	non-invasive		
Thio et al. [102]	×			
Rosenberg et al. [85]	×			
Repp et al. [83]	×			
QOSH [9]	×			
Boniface et al. [13]	×			
Cao et al. [15]	No technical details			
Jconsole [47]			×	
Ganglia [62]				×
Chukwa [82]				×
Nagios [74]				×
PCMONS [28]				×
DARGOS [79]				×
RMCM [91]	×		×	×

As shown in Table 3.1, existing monitoring approaches, operating at the software layer are invasive. They require access to the source code of the services being monitored and are typically operated by the provider. These approaches can not be easily applied in the context of Cloud computing, since they modify the source code of the client and/or the service. Therefore, it is necessary to define a non-invasive monitoring approach for Cloud services. The target approach

does not require access to the source code of a service, and can be installed by the client. It neither modifies the server implementation, nor the client code.

Table 3.1 has also shown that the majority of existing monitoring approaches do not deal with all Cloud layers, except RMCM [91], which still lacks some flexibility in adjusting the monitored parameters. Thus, it is crucial to propose a multi-layer monitoring approach that operates at all Cloud layers and collects all related parameters.

The next section presents and discusses existing analysis approaches for Cloud computing environments.

3.3 Analysis

Monitoring is essential for analyzing a Cloud computing environment. Therefore, analysis approaches are naturally based on the use of existing / new monitoring tools. In this section, we present and discuss analysis approaches for Cloud computing environments and their related monitoring tools. Existing analysis approaches can be classified according to three main criteria:

- The Cloud layer
- The architectural design (centralized or distributed)
- The need of storage space

We use the same criteria as in Section 3.2 to classify the used monitoring approaches.

3.3.1 Centralized Analysis Approaches

Teixeira et al. [101] propose a monitoring and analysis architecture for data centers. The proposed architecture is called HOLMES. It is based on the publish-subscribe paradigm, where all principal modules are viewed as publishers and subscribers to an event broker (i.e., event/message bus). HOLMES consists of four modules. The first one is a CEP engine. It is in charge of correlating monitoring events, via user-defined CEP queries. The second module is based on machine learning techniques. It analyses incoming monitoring data, organized in time series, to detect performance anomalies. The third module is called "visualization". It feeds real-time web charts. The last module is called "storage". It is used for storing historical data. HOLMES is based on a single CEP engine. The used CEP engine executes analysis queries to detect performance anomalies. These queries get the observed data of the monitoring sensors as inputs, process them and generate alarms when an anomaly is detected. The used monitoring sensors act at the infrastructure layer. This means that HOLMES is not able

to detect anomalies related to higher layers. Moreover, HOLMES uses straightforward analysis rules that analyze only a single category of parameters. The proposed analysis rules are mostly based on pre-defined thresholds. Thus, this analysis is not able to accurately identify the source of an anomaly due to the lack of relevant analyzed metrics. The architecture of HOLMES is centralized, which suffers from the bottleneck and single point of failure issues in the case of large scale systems.

Narayanan et al. [75] define a monitoring and analysis framework for data centers. The proposed framework is based on CEP techniques. It consists of three main components. The first component is in charge of monitoring the data centers. In particular, it monitors the storage, servers, network and applications. The second component, called Event Collector Engine, allows to (1) collect the monitored data, (2) convert collected data into a special format and (3) send it to the CEP engine, which is the third component of the monitoring framework. The CEP engine correlates the incoming events via the execution of a set of CEP queries. The open source system CAYUGA [14] has been used to define the used CEP queries.

The proposed framework [75] utilizes a single CEP engine. Therefore, it suffers from the bottleneck issue in the context of large scale Cloud environments.

C-Meter [109] is a performance analysis framework for Cloud computing. It is an extension of Grenchmark: a Grid tool, allowing performance tests in Grid computing environments. C-Meter is composed of three subsystems: (1) *Core* subsystem, (2) *Cloud Interaction* subsystem and (3) *Utilities* subsystem. The Core subsystem provides the main functions of C-Meter. It is composed of three modules. The first one, called "Listener", listens to job submissions from the workload generator of GrenckMark and commands users (e.g., terminate an experiment). The received job descriptions are queued to the second module: Job Queue. Jobs remain in the Job Queue until they become ready for submitting. The third module is called "Job Submission". It allows to transfer the job to the "execution" module, which executes the job on virtual resources (i.e., the Cloud). The Execution modules reports statistics to C-Meter. The Cloud Interaction subsystem is in charge of managing the interactions between C-Meter and the Cloud environment under test. It is composed of two modules: (1) the Resource Management module and (2) the Connection Management module. The Resource Management module is used to allocate and release virtual machines / resources. The Connection Management module is in charge of establishing connections with the Cloud environment. The Utilities subsystem provides basic utilities such as configuring the test and providing statistic funtionalities. As mentioned above, C-Meter makes use of the workload generator of GrenchMark. It generates synthetic and real workloads. These workloads, consisting of sev-

eral jobs, are submitted to analyze a Cloud's performance. The studied analysis framework mainly operates at the infrastructure layer, since it monitors virtual resources. The authors are aware of the necessity of monitoring jobs, since this would improve the results of the analysis.

Ostermann et al. [76] evaluate the performance of Amazon's EC2 Cloud. The authors launch large scale experiments on EC2, an IaaS Cloud. The presented performance assessment is mainly based on monitoring selected performance-related parameters, but does not consider relations between these parameters.

Cohen and Chase [21] present an analysis approach for web services architectures consisting of three layers: Web Server (Apache), AppServer (BEA Weblogic) and Database Server (Oracle). The proposed analysis approach is based on the use of Tree-Augmented Bayesian Networks (TAN). The used TANs model the correlations between the Service Level Objective (SLO) and the resource usage. The proposal allows (1) the detection of SLO violations via simple comparisons with pre-defined thresholds; and (2) the identification of the cause of a violation via the use of correlations between the different monitored metrics. The TANs (modeling correlations) are deduced from historical data that were previously collected and stored in a separate database. This means that a training period is necessary for the execution of this approach. During this training period, the system is unable to identify the cause of a violation. Moreover, it is hard to apply this approach in the context of Cloud computing due to the large volume of data that has to be stored.

EbAT [104] is an online detection method of Cloud computing anomalies. It analyzes distributions of metrics instead of individual metrics. EbAT follows three main steps to calculate and analyze the multi-level entropy. First, EbAT collects Cloud metrics. Cloud components are organized in a hierarchy and clustered into two groups: leaf node and non-leaf node. A leaf node gathers data form its local sensors, while a non-leaf node collects data from its local sensors and its child nodes. Afterwards, it normalizes and groups data into intervals. According to the type of the node (i.e. leaf or non-leaf node), monitoring events are either generated from "local metrics" or from "local metrics and child nodes". The generated monitoring events are, then, used to calculate the entropy time series. Finally, EbAT analyzes the calculated entropy time series via the use of spike detection methods or signal processing approaches of subspace solutions. The used spike detection methods are: (1) visual identification and (2) exponential weighed moving average (EWMA), while the used signal processing approach is wavelet analysis. EbAT is based on the use of online singular value decomposition (SVD) as a subspace analysis approach.

EbAT collects OS (Operating System), applications and platform metrics. It

does not take into account infrastructure metrics for the analysis. Moreover, it needs a storage space to keep historical data.

Li et al. [57] propose a CEP-based monitoring system for Cloud computing environments. The defined system is composed of four main components: (1) the basic event cloud, (2) the CEP engine, (3) the action part and (4) the database. The basic event Cloud is used to monitor Cloud computing systems. The CEP engine, based on the database, implements the detection algorithm and triggers alarms when an unhealthy situation is detected. The action part is associated to a dashboard, which allows a real-time visualization of monitoring data. The CEP engine is based on a CEP-rule setting interface, which allows users to define and launch CEP queries. The detection algorithm is the core component of the CEP engine. It analyzes recorded events and detects unhealthy situations via the use of statistical functions.

The proposed system operates at the software layer and is based on a single CEP engine that might become a bottleneck in the context of large scale Cloud environments.

Zhu et al. [111] propose a fault diagnosis framework for Cloud-based systems. The proposed framework is composed of two main components: (1) the offline model training and (2) the online fault diagnosis. First, the offline model training is in charge of generating training data for applications with injected faults. Second, it monitors the Cloud (i.e. applications, virtual machines, physical machines and clusters). The monitoring data is, then, used by the offline model training to create the substitution graph and the detection graph. The substitution graph is based on the recipe concept by replacing a metric A (ingredient A) by metric B (ingredient B). Actually, it examines the correlation between metrics/events and classifies the metrics/events into clusters. Each cluster includes strongly correlated metrics/events. Therefore, the substitution group allows to substitute one event/metric by another one, belonging to the same cluster. The normalized Mutual Information (NMI) is used to identify these clusters. In fact, NMI is considered as a similarity measure. The detection graph allows to detect the fault. Zhu et al. defines the fault pattern as an ordered sequence of events. They use the EdgeRank [30] algorithm to identify the most critical events that should be included in the fault pattern. The online fault diagnosis uses the learned fault patterns and constructed graphs to generate the diagnosis paths. The generated diagnosis paths are used to diagnose the fault and identify its cause.

This proposal operates at the virtual infrastructure and software layers; and follows a centralized design. Moreover, it does not exploit the relationships between Cloud metrics to optimize the defined fault patterns and diagnosis paths.

3.3.2 Distributed Analysis Approaches

Leitner et al. [56] propose a monitoring and analysis approach for Cloud applications. Their approach is integrated within the "CloudScale" framework, and allows to track the application performance. The presented approach is based on the concepts of event-based monitoring and complex event processing. So, it mainly consists of event emitters and CEP engines. The event emitters monitor Clouds and emit monitoring events. The produced events are processed by the used CEP engines to detect performance-related problems. This approach makes use of several CEP engines organized in a hierarchical fashion. The authors define three correlation levels that are processed by three different CEP engines. The first level concerns "host correlation", and it is realized by the first involved CEP engine. Its results are sent to the second CEP engine that accomplishes "resource correlation". The results of the "resource correlation" CEP engine are forwarded to a third CEP engine that is in charge of "application correlation".

This approach needs a database for storing metrics. It generates considerable network traffic, due to the large number of messages exchanged between the different CEP engines, as shown by the experiments performed by the authors.

Baresi and Guinea [7] propose an event correlation middleware, called ECoWare. It is a distributed event correlation and aggregation framework for multi-layer monitoring. ECoWare consists of three main components: (1) the probes, (2) the processors and (3) the ECoWare dashboard. The probes are the used monitoring tools. They are inserted into the execution environments to collect relevant metrics. For instance, the probes inserted into Ubuntu virtual machines are based on the collectd tool [22]. The processors are used to process monitoring data. They can be clustered into three groups: (i) KPI processors, (ii) Aggregators and (iii) Analyzers. The KPI processors are used to implement mlCCL KPI collection. mlCCL (multi-layer Collection and Constraint Language) is an event processing language. The Aggregators are in charge of data aggregation. The KPI processors and aggregators are built using the CEP Esper [96]. The Analyzers allows to analyze data.

ECoWare uses two cooperating CEP engines. It consists of several components that communicate via a publish/subscribe system (Siena). In a large scale Cloud infrastructure, it is likely that the number of messages exchanged between these components in ECoWare will increase significantly.

Kutare et al. [55] propose a large scale monitoring and analysis approach, called Monalytics, to manage large scale data centers. Monalytics allows to monitor and analyze Xen-based environments. It includes a set of agents and brokers. The agents are used to monitor the system. The brokers allow to aggregate and analyze the outputs of the agents (i.e., monitoring data). Monalytics is based on the use of the EVPath eventing system to enable the communication between the

agents and the brokers. The analysis approach EbAT [104] is used by Monalytics to detect a problem. Monalytics focuses on the monitoring of a Cloud infrastructure (Infrastructure-as-a-Service (IaaS)) and virtualized systems (i.e., Xen-based systems). The authors define different deployment topologies between monitoring brokers and analysis agents. The topologies are represented as graphs and can be managed at runtime. The Monalytics system monitors CPU and memory utilization. It is integrated into the Xen virtualization infrastructure.

Monalytics needs a database for storing data. It is limited to the IaaS layer and does not take into account monitoring data from higher layers (i.e., Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) layers). The communication between the brokers and agents could introduce a high network traffic.

Haibo et al. [72] propose a performance analysis approach for Cloud computing systems, called Magnifier. It allows to detect performance-related problems, while identifying their prime causes. Magnifier includes a monitoring and an analysis modules. The monitoring module is based on trace collecting daemons, which are installed on each server. The used daemons collect traces related to requests. The authors propose a hierarchical structure (i.e. tree) to model a request. The proposed hierarchical structure is composed of three layers (component, module, function). This structure helps the authors to diagnose Cloud services, detect performance-related problems layer by layer, separately and identify their root causes. The detection procedure is based on a basic threshold comparison. Magnifier makes use of a diagnosis algorithm to localize abnormal elements (e.g. components, modules and functions) at each layer. The proposed diagnosis algorithm follows 3 steps. First, it calculates the normal and abnormal latency of each element and their corresponding fluctuation range. Then, it calculates the fluctuation extent of each element. Finally, elements are ranked according to their extent. This ranking allows to identify abnormal elements.

Magnifier operates at the software layer and does not consider the effect of the platform and infrastructure layers. Monitored metrics have to be stored in a database.

Dyk [29] presents a distributed monitoring and analysis architecture based on CEP engines. He illustrates the disadvantages of such architectures, in particular that distributed CEP architectures suffer in general from a synchronization problem.

Grell and Nano [40] discuss issues related to large scale Internet services monitoring via CEP. They propose two different CEP architectures, a centralized and a distributed CEP architecture.

The proposed centralized architecture consists of four components. The first one allows us to monitor services. It is based on the use of a Watch Dog in-

frastructure that generates syntactic transactions. The second component is a publish-subscribe system. It forwards the results of the syntactic transactions to the third component of the architecture, called SLA monitoring service. The latter analyzes the received data and returns the state of the monitored service. The analysis is mainly based on threshold comparison. The results of the SLA monitoring service are sent back to the publish-subscribe system, which forwards them to the e-mail service, fourth component of this architecture, in case a SLA violation has been detected. The proposed distributed architecture is similar to the centralized one. The main difference is the fact that it includes one SLA monitoring service per machine.

The work indicates that the choice of the CEP architecture depends on the requirements of the monitored system. The approach is not useful for Cloud monitoring and analysis, due to the dynamicity of a Cloud environment. Moreover, the proposed architectures operate at the software layer and do not consider the platform and infrastructure layers.

Balis et al. [6] propose a CEP-based approach for real-time grid monitoring and analysis. The proposed approach is called GEMINI2. It is based on a distributed CEP architecture. GEMINI2 makes use of three main components: sensors, monitors and clients. Each sensor incorporates a CEP engine. It generates simple events that describe the state of grid components. The sensors allow to reduce the number of events via the use of reduction rules. The monitors are based on CEP engines. They run the main analysis rules. The clients generate complex events by subscribing to monitors.

GEMINI2 may overload the virtual machines, since it assigns a CEP engine to every sensor. Moreover, it could introduce a high network traffic in the context of Cloud computing environments. Furthermore, GEMINI2 operates at the infrastructure layer and does exploit the interactions with higher layers to detect the problem.

3.3.3 Discussion

Table 3.2 summarizes the studied analysis approaches, while focusing on their main characteristics.

Table 3.2: Analysis approaches for Cloud computing environments[1]

Analysis Approaches	Monitoring					Analysis								
	S		P	VI	PI	Centralized				Distributed				Storage
	Invasive	Non-invasive				S	P	VI	PI	S	P	VI	PI	
HOLMES [101]				×				×						needed
Narayanan et al. [75]	×				×	×			×					
C-Meter [109]				×				×						
Ostermann et al. [76]				×				×						needed
Cohen et al. [21]	×		×		×	×	×		×					needed
EbAT [104]	×		×	×		×	×	×						needed
Li et al. [57]	Not mentioned					×								needed
Zhu et al. [111]	Not mentioned			×	×	×		×	×					needed
Leitner et al. [56]	×		×	×						×	×	×		needed
ECoWare [7]	Not mentioned			×	×					×	×	×		
Monalytics [55]				×								×		needed
Magnifier [72]	×									×				needed
GEMINI2 [40]												×		

[1] **S:** Software layer; **P:** Platform layer; **VI:** Virtualization layer; **PI:** Physical Infrastructure layer

As shown in Table 3.2, the majority of existing approaches do not exploit the relationships between Cloud metrics, and need storage space to record monitored events. Thus, it is necessary to propose an analysis approach that takes into account the interactions between Cloud layers. The target approach should be able to process events without storing them.

Moreover, Table 3.2 shows that existing analysis approaches are based on either a centralized architecture or a distributed one. Each architecture has some disadvantages. In existing approaches based on a single CEP engine, the CEP engine itself represents a single point of failure. Moreover, a single CEP engine could easily become a bottleneck if the amount of monitored data exceeds the processing capacities of the CEP engine. On the other hand, distributed CEP architectures suffer from the potentially large number of messages exchanged between the distributed CEP engines. Therefore, it is crucial to design a dynamic architecture that dynamically switches between different architectures depending on the current conditions of the observed Cloud environment.

As mentioned above, analysis approaches allow us to characterize the state of a Cloud computing environment and detect unwanted situations. Their outputs are usually used by the last step of the self-healing process (i.e. recovery) to fix detected problems. Section 3.4 presents and discusses existing recovery and self-healing approaches for Cloud computing environments, in the presence of performance issues.

3.4 Recovery / Self-healing Approaches for Cloud Computing Environments

Monitoring and analysis are important to detect and recover performance-related problems, which might occur in Cloud computing environments. Thus, existing recovery approaches include monitoring and analysis components. All together (i.e., monitoring, analysis and recovery) constitute a self-healing solution. In this section, we present, discuss and classify self-healing approaches for Cloud computing environments, in the presence of performance issues. The classification of recovery approaches is based on the two following criteria:

- The addressed Cloud layer
- The capability of the recovery approach in validating the applied action

We use the same criteria as in Section 3.2 and 3.3 to classify the used monitoring and analysis approaches, respectively.

FDCS [11] is a fault detection framework for Cloud systems. It makes use of Ganglia to monitor virtual machines. Monitored data are stored in a database. In FDCS, Cloud machines collaboratively monitor performance parameters of

other machines in the system and trigger an alarm, if a fault occurs. They are organized in a unidirectional ring (i.e., a machine is able to communicate with another machine having a higher id). FDCS is based on a distributed outlier detector algorithm to detect the failed machine and the responsible parameter (e.g., CPU). The proposed outlier detector algorithm is faster than centralized approaches, since it is based on in-network processing of messages. FDCS is based on a distributed setup with a central machine for final reporting, called reporter. The reporter machine reports the activities of all outlier detectors. FDCS's workers have two operation modes: push and pull. FDCS allows us to isolate the fault and identify the most faulty features. This avoids cascading faults. However, this does not repair the fault.

FDCS operates at the virtualization layer and does not exploit the relationships between Cloud metrics. Therefore, it is not able to accurately characterize the failure and its cause.

Dai et al. [25] propose a self-healing approach for Cloud computing environments. The proposed approach is based on consequence-oriented diagnosis. It uses the observed symptoms to predict the consequences. The proposal combines multiple-valued decision diagrams and the Naive Bayes Classifier to determine the severity level of the system, identify the consequences and prevent them. Based on the multiple-valued decision diagram and the observed symptoms, the proposed system defines the severity level of the observed Cloud situation. Second, it identifies the consequence category that corresponds to the defined severity level. Afterwards, the system identifies the required diagnosis system, according to the identified category. Actually, a diagnosis system is assigned to each category of consequence. This step is based on a "Naive Bayes Classifier" approach. The identified diagnosis system allows to open and execute the most suitable prescription for healing. The results of healing are, then, sent back to the system to update the Naive Bayes Classifier in the diagnosis system. This step is based on a set of machine learning techniques and allows to improve the quality of recovery.

The approach proposed by Dai et al. monitors the virtualization layer and stores collected parameters (i.e. symptoms) in a database. It only addresses performance degradations occurring at the virtual infrastructure layer.

Magalhaes et al [60] propose a self-healing framework for Cloud-host web-based applications, called SHoWA. It allows Cloud customers to launch recovery actions when a performance degradation occurs. SHoWA consists of three main components: the "Monitoring" component, the "Data analysis" component and the "Recovery" component. The Monitoring component is composed of two main modules: the "Sensor" and the "Data preparation" modules. The Sensor module collects application, application server and system metrics. The data collected by the Sensor module of SHoWA can be classified into two groups, according to

their level of granularity. The first group includes "user-transaction level monitoring" data (i.e., system and server application metrics), such as CPU load and JVM heap memory. The second group includes "profiling level monitoring" data (i.e., application metrics). The Data preparation module is in charge of aggregating the data gathered by the Sensor module. The diagnosis module of SHoWA examines the correlation coefficient value between 2 parameters belonging to 2 different Cloud layers (virtual infrastructure and software layers), to detect the degradation and identify its cause. The diagnosis module is running on a single VM. It has to store all monitored data and calculate the correlation coefficient between every pair of metrics. Therefore, the decision could take a lot of time in the context of large scale Cloud computing environments.

The recovery module of SHoWA is similar to a disease database. It is based on the use of "If-Then" rules to fix detected performance-related problems. Moreover, it does not validate the applied recovery action. SHoWA does not monitor the physical infrastructure layer.

Alhosban et al. [1] propose a self-healing technique for Cloud services (software layer), called a fault occurrence likelihood estimation and exception technique. It allows to monitor, evaluate performance parameters of Cloud services and repair services in case a fault has been detected. The proposed technique starts by collecting service parameters such as service history, execution time and other QoS parameters. Afterwards, it analyzes collected data by calculating the fault likelihood of the services. This allows to identify and generate the adequate planning strategies. The generated planning strategies are then stored for future use. To repair a service, the proposed technique follows two steps. First, it chooses the most adequate recovery action. Second, it executes it. To achieve the first step (i.e., choose the best recovery action), the proposed technique computes the fault likelihood and the utility of the service.

The technique proposed by Alhosban et al. operates at the software layer, does not consider the relationships between Cloud layers and does not check the success of the applied recovery action. Thus, it is not able to accurately characterize the failure and efficiently repair it.

Casalicchio et al. [16] propose a self-adaptable architecture for Cloud-based systems. It allows to manage resources under changing conditions. The proposed architecture is modular and based on the MAPE-K loop. Therefore, it consists of 5 modules. The "Monitor" module is in charge of collecting and aggregating (1) performance indexes such as CPU utilization and network traffic and (2) workload details like arrival time and average response time. The "Analyze" module diagnoses monitoring data to decide whether an adaptive action is needed. The conducted analysis is based on the requested level of QoS. In case an adaptive action is required, the "Plan" module is triggered. Based on the state of the Cloud

system and its performance model, the "Plan" module identifies the adequate adaptive action (allocate or de-allocate VMs). The chosen adaptive action is then applied by the "Execute" module. The "Knowledge" module stores information about the Cloud system and is updated during the stages of the autonomic loop.

The "Analyze" module of this approach is based on comparing the values of monitored parameters to pre-defined thresholds. It does not exploit relationships between Cloud layers. The "Plan" module is quite simple and only considers two adaptive actions to solve the problem. Moreover, it does not check the success of the applied adaptive action. This proposal does not consider platform and physical infrastructure layers.

Sarkar et al. [89] present a self-healing system for PaaS Clouds to deal with performance-related problems. It is based on a layered architecture. The first layer, called "monitoring layer", includes the used monitoring tools that gather infrastructure and platform metrics. The second layer is called "automated incident management layer". It is composed of a centralized database and 6 components: (1) the event aggregation and correlation system, (2) the automated incident management system (AIMS), (3) the incident ticket creation and resolution, (4) the automated incident handling, (5) the workflow system and (6) the provisioning system. The analysis is based on the two first components: the Event Aggregation and Correlation System and the Automated Incident Management System (AIMS). The Event Aggregation and Correlation System is in charge of receiving events from the monitoring module. It stores received events within a window of time. Then, it performs aggregation, correlation and suppression of events. Subsequently, a filtering activity is performed. It decides about events that should be sent to the AIMS. The latter detects performance incidents. Correlations, in this work, are determined by the aggregation operations. These just decide about the relevant events to observe and do not aim to extract relationships between collected metrics. This is the reason why this analysis approach cannot accurately identify the cause of a performance incident. Moreover, this approach only acts on the platform layer. The recovery is based on the four last components. The "Automated Incident Handling" is used to apply a corrective action, if an incident has been detected. It communicates with the "Workflow System" to schedule a workflow. The "Incident Ticket Creation and Resolution" is used to create and resolve a ticket, when the event should be ignored (no incident has happened).

Discussion

Table 3.3 summarizes the studied self-healing solutions. It shows the main characteristics of the involved monitoring, analysis and recovery approaches.

Table 3.3: Self-healing approaches for Cloud computing environments[2]

Self-healing Approaches	Monitoring					Analysis									Recovery			
	S		P	VI	PI	Centralized				Distributed				Storage	Success validation			
	Invasive	Non-invasive				S	P	VI	PI	S	P	VI	PI		S	P	VI	PI
FDCS [11]				×								×		needed			N	
Dai et al. [25]				×				×						needed			Y	
SHoWA [60]		×	×	×		×	×	×						needed	N	N	N	
Alhosban et al. [1]	Not mentioned					×								needed	N			
Casalicchio et al. [16]	Not mentioned			×		×		×						needed			N	
Sarkar et al. [89]			×				×							needed		N		

[2]**Success validation:** Does the recovery approach check the success of the applied action? **Y:** Yes; **N:** No

As shown in Table 3.3, existing self-healing approaches do not deal with all Cloud layers. They typically focus on one of the stages of the self-repairing process (monitoring and/or analysis) and define simple recovery approaches. The majority of existing recovery approaches do not check the success of the applied action. Therefore, it is crucial to define a cross-layer reactive monitoring approach that **monitors**, **detects** and **rectifies** performance-related problems, which might occur on **all Cloud layers**.

The conducted study and discussion of related approaches (monitoring, analysis and recovery) have allowed us to identify the requirements of this PhD thesis. They are summarized in the next section.

3.5 Requirements Catalog

This thesis should fulfill the following requirements:

R1: It is necessary to propose, design and implement a non-invasive approach for Cloud services (software layer).

R2: It is required to propose and validate a multi-layer monitoring approach for Cloud computing environments that collects performance parameters belonging to all Cloud layers.

R3: It is crucial to define and validate a cross-layer analysis approach for Clouds, which exploits the relationships between Cloud layers to rapidly detect a performance-related problem and accurately identify its cause.

R4: It is crucial to define and validate a dynamic architecture for Cloud analysis, to ensure the scalability of the analysis agent and fits to the elasticity property of Clouds (i.e., scale up/down).

R5: It is necessary to define and validate a multi-level recovery approach that applies actions on all Cloud layers and validates the success of the applied recovery action.

R6: It is crucial to combine the proposed approaches (Monitoring, Analysis and Recovery) in a single framework, to fix performance-related problems that might occur in Cloud computing environments.

3.6 Summary

This chapter has analyzed and discussed related work (monitoring, analysis and recovery approaches). Based on this analysis, we have identified the requirements that should be fulfilled by this thesis. The next chapter details the proposed approach.

"I want to do it because I want to do it."

Amelia Earhart

4

CEP4Cloud: Complex Event Processing for Reactive Cloud Monitoring

4.1 Introduction

This chapter presents our cross-layer reactive performance monitoring approach for Cloud computing environments, called CEP4Cloud. It is based on the methodology of Complex Event Processing and allows us to detect performance-related problems and fix them with minimal human intervention. First, we give an overview of CEP4Cloud, while outlining its general architecture (see Section 4.2). Then, we describe the main components of CEP4Cloud. The multi-layer monitoring agent is detailed in Section 4.3. Section 4.4 describes the cross-layer CEP-based analysis agent. The action manager framework is presented in Section 4.5. The last section summarizes this chapter. Parts of this chapter have already been published in [64–67, 69].

4.2 CEP4Cloud in a Nutshell

This section presents our reactive performance monitoring approach for Cloud computing environments, called CEP4Cloud. First, we outline its general architecture. Then, we briefly describe the main components of CEP4Cloud.

4.2.1 The Architecture

Figure 4.1 shows the architecture of CEP4Cloud. It consists of three main components:

- The multi-layer monitoring agent
- The cross-layer CEP-based analysis agent
- The action manager framework

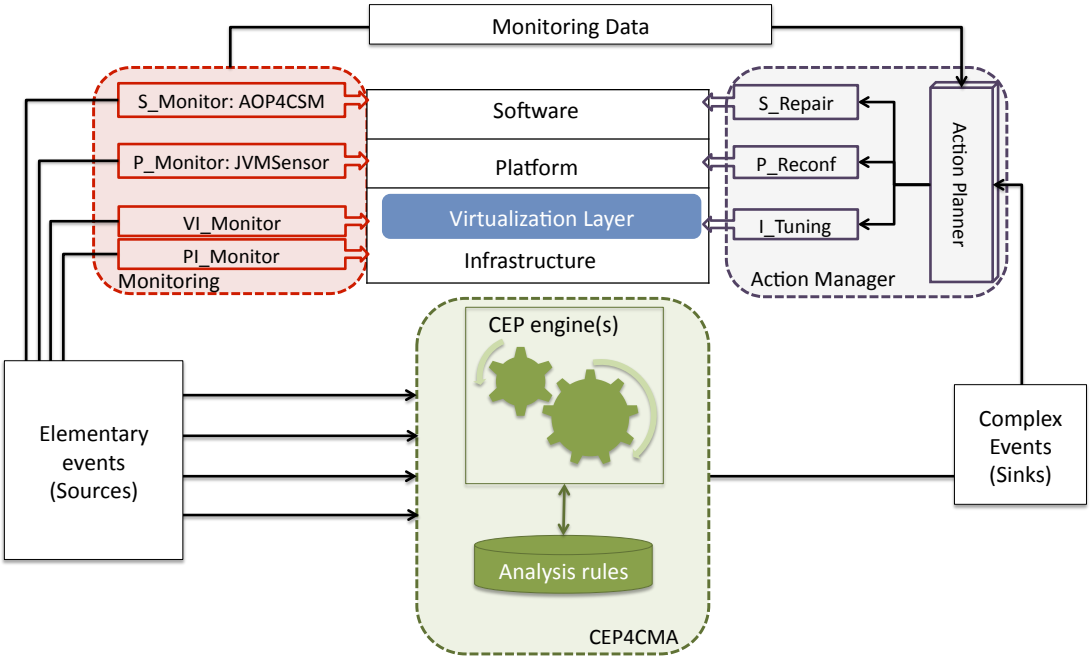

Figure 4.1: The architecture of CEP4Cloud

The multi-layer monitoring agent relies on four monitoring components:

- PI_Monitor: Physical Infrastructure Monitor
- VI_Monitor: Virtual Infrastructure Monitor
- P_Monitor: Platform Monitor
- S_Monitor: Software Monitor

PI_Monitor, VI_Monitor, P_Monitor and S_Monitor send collected data to the analysis agent of CEP4Cloud. It is based on the use of a CEP engine to process and analyze received events. The conducted analysis detects performance-related problems, generates diagnosis reports and sends notifications to the action

manager framework, if a performance degradation occurs. Based on the received diagnosis reports, the action manager framework identifies the adequate recovery action, executes it and validates it.

The multi-layer monitoring agent, the analysis agent and the action manager framework are described in Sections 4.3, 4.4 and 4.5, respectively and briefly introduced below.

4.2.2 The Monitoring Agent

The monitoring agent consists of 4 components. They make use of existing sensors, as discussed below.

- **PI_Monitor** monitors the physical infrastructure layer and gathers metrics related to the consumption of hardware resources, such as memory consumption. PI_Monitor uses a set of existing sensors: Xenmon [41], Ganglia [36, 62], IoStat [45] and MpStat [73].
- **VI_Monitor** monitors virtual machines. It uses Ganglia, MpStat and IoStat.
- **P_Monitor** operates on the platform layer and gathers corresponding data.
- **S_Monitor** operates on the software layer and collects performance parameters of Cloud services at the SaaS layer, such as the execution and communication times. It is based on the use of AOP4CSM, a monitoring approach based on aspect-oriented programming proposed and validated in the context of this thesis [69].

4.2.3 The Analysis Agent

The analysis agent of CEP4Cloud allows to analyze recorded events and detect performance-related problems, while identifying their causes. The analysis agent is based on CEP. It makes use of a set of cross-layer analysis rules. They are implemented in the Esper CEP engine, as queries. The used analysis rules are defined on the basis of relationships between the monitored parameters across Cloud layers, while following a root cause analysis approach.

4.2.4 The Action Manager Framework

The action manager framework is used to fix performance-related problems that might occur in Cloud computing environments. It consists of two main components: the action planner and the action executor. The action planner allows to identify and validate the suitable recovery action. The action executor operates at all Cloud layers and executes the identified recovery action.

The main components of CEP4Cloud are based on new monitoring, analysis and recovery approaches. They are described in Sections 4.3, 4.4 and 4.5, respectively.

4.3 Monitoring

The multi-layer monitoring agent is based on the use of (1) our novel Cloud service monitoring approach (AOP4CSM); and (2) the results of existing monitoring tools that typically operate on a single Cloud layer (e.g., [28, 62, 82]). Thus, the four components of our monitoring agent described above make use of AOP4CSM and existing sensors, as presented below.

4.3.1 PI_Monitor

PI_Monitor monitors the physical infrastructure layer and gathers metrics related to the consumption of hardware resources, such as the CPU usage of the physical cores, the waiting and blocked times spent by virtual machines to access the physical disk, the memory consumption etc. It is installed in the privileged domain (Dom0) with direct access to the XEN hypervisor, the virtualization technology used in this work. PI_Monitor makes use of a set of existing sensors: Ganglia [36, 62], IoStat [45], MpStat [73] and Xenmon [41]. Xenmon collects information about the CPU such as the blocked, the waiting times and the number of executions per second. Ganglia mainly gathers information about the state of resources, such as disk and memory consumption. Moreover, it measures the network link quality. IoStat measures disk transactions such as the I/O requests to the physical disk. It is running in the privileged domain (Dom0) that has direct access to the physical disk. Finally, MpStat gathers the software and hardware interrupts of Dom0. Table 4.1 details the metrics collected by PI_Monitor.

4.3.2 VI_Monitor

VI_Monitor operates at the virtualization layer and monitors virtual machines. It is installed on each virtual machine. VI_Monitor makes use of Ganglia, MpStat and IoStat. Ganglia collects CPU, RAM, disk and network metrics of the virtual machines, such as the used swap and the throughput of the network (BytesIn and BytesOut). MpStat measures the CPU steal of virtual machines, reflecting the time spent by the VM waiting for the hypervisor's tasks. IoStat gathers information about the number of read and written pages of the considered virtual machine (see Table 4.2).

Table 4.1: PI_Monitor: components, used sensors and monitored metrics

Used Sensors	Metrics	Metrics' Designation
Xenmon	CpuUsedDom0	CPU used by the privileged domain (Dom0)
	CpuBlockedDomU	The time spent by DomU, blocked on the physical CPU
	CpuWaitingDomU	The time spent by DomU, waiting on the physical CPU
	ExecSec	The number of executions per seconds
Ganglia	P-DiskFree	The free disk value
	P-DiskUsed	The used disk value
	P-RamFree	The free memory value
	P-RamUsed	The used memory value
	P-SwapFree	The free swap value
	P-RamUsed	The used swap value
	P-Load	The load of Dom 0
	P-ProcNb	The number of running processes
	P-BytesIn	The number of bytes in
	P-BytesOut	The number of bytes out
IoStat	IOReqDisk	The number of I/O requests to the hard disk
	P-Read	The number of read pages
	P-Wrtn	The number of written pages
MpStat	HwInterr	The number of hardware interrupts
	SwInterr	The number of software interrupts

4.3.3 P_Monitor

P_Monitor operates on the platform layer and gathers corresponding data. For this purpose, the JVMSensor tool has been developed to deal with Java Virtual Machine (JVM) monitoring. JVMSensor measures JVM platform metrics, such as the CPU time of the running threads, the heap memory and the number of loaded classes. Table 4.3 shows all metrics gathered by JVMSensor. JVMSensor is based on the Jconsole tool [47].

4.3.4 S_Monitor (AOP4CSM)

S_Monitor operates on the software layer and collects performance-related parameters of Cloud services at the SaaS layer. It makes use of our monitoring approach, called AOP4CSM for "Aspect-Oriented Programming for Cloud Service Monitoring". It is detailed below.

Table 4.2: VI_Monitor: components, used sensors and monitored metrics

Used Sensors	Metrics	Metrics' Designation
Ganglia	CPUuser	The CPU user of the virtual machine (VM)
	CPUSystem	The CPU System of the VM
	CPUWait	The CPU Wait of the VM
	V-DiskFree	The free disk value of the VM
	V-DiskUsed	The used disk value of the VM
	V-RamFree	The free memory value of the VM
	V-RamUsed	The used memory value of the VM
	V-SwapFree	The free swap value of the VM
	V-RamUsed	The used swap value of the VM
	V-Load	The load of the VM
	V-ProcNb	The number of running processes of the VM
	V-BytesIn	The number of bytes in of the VM
	V-BytesOut	The number of bytes out of the VM
IoStat	V-Read	The number of read pages of the VM
	V-Wrtn	The number of written pages of the VM
MpStat	CpuStealVM	The time spent by the virtual machine, waiting for the hypervisor tasks

Table 4.3: P_Monitor: components, used sensors and monitored metrics

Used Sensors	Metrics	Metrics' Designation
JVMSensor	CpuTimeTh	The CPU time of a running thread
	WaitedCountTime	The number of times, the thread was waiting to access the CPU
	ExcepNb	The number of handled exception per second
	HeapMem	The heap memory usage of the JVM
	NonHeapMem	The Non heap memory usage of the JVM
	ClassNb	The number of loaded classes

AOP4CSM is a monitoring approach based on aspect-oriented programming. It measures five QoS parameters: (i) execution time, (ii) response time, (iii) communication time, (iv) throughput and (v) availability. They are explained below.

- The **response time** T_{resp} defines the time needed to serve a request [85]. It starts when the client sends its request and finishes when the client receives the corresponding response. Thus, it is the temporal difference between the

instant (i_1) when the client invokes the request and the instant (i_8) when the client receives the corresponding response (see Figure 4.2).

- The **execution time** T_{exec} measures the time needed to execute a request on the server (see Figure 4.2) [10].
- The **communication time** T_{com} is the time needed to transfer the request from the client to the server plus the time needed to transfer the response from the server to the client (see Figure 4.2). Thus, the communication time is the response time minus the time necessary for executing the request and the time necessary for processing the messages involved (see Equation 4.20).

$$T_{com} = T_{resp} - (T_{exec} + T_{processing}) \tag{4.1}$$

$T_{processing}$ is the time required for message processing (see Figure 4.2).

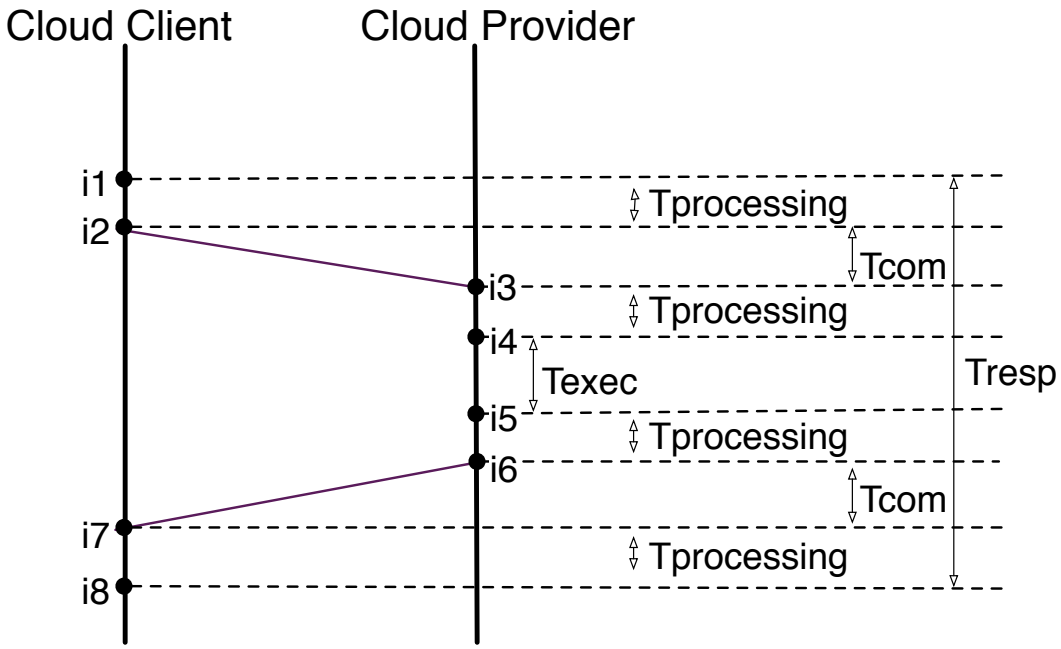

Figure 4.2: QoS parameters

When using Axis, $T_{processing}$ corresponds to the time consumed by the Axis handler chain to perform message processing (such as dealing with security attributes or adding a message header for reliable messaging). In this work, $T_{processing}$ is a negligible value, since the handler chain is typically empty.

- The **throughput** measures the number of successful requests (SuccRequests) during a period of time T (see Equation 4.2) [10].

$$Throughput = SuccRequests/T \tag{4.2}$$

SuccRequests represents the number of successful requests during a period T. T is a parameter that is set when AOP4CSM is configured. A successful request is a request with a successful response; a successful response is a response that successfully reached the server.

- The **availability** measures the accessibility of a service [10]. It is calculated using the formula shown in Equation (4.3).

$$Availability = SuccRequests/AllRequests \tag{4.3}$$

AllRequests is the number of all requests sent during the period T.

AOP4CSM gathers and assesses the QoS parameters values described above. Its approach of measuring these parameters is described below.

Functionality of AOP4CSM AOP4CSM is based on aspect-oriented programming code that intercepts client and server methods at well defined join points to collect data at important instants of time. These instants are t_1, t_2, t_3 and t_4 (see Figure 4.3); where

- t_1 is the instant when the client invokes the request
- t_2 is the instant when the server receives the request
- t_3 is the instant when the server sends the response
- t_4 is the instant when the client receives the response

In addition, the proposed aspect code computes the number of successful invocations (processed requests) while intercepting the corresponding method at the join point that corresponds to t_4. Moreover, the number of all sent requests is evaluated by advice 1 (see Figure 4.3).

Recorded timestamps allow us to assess QoS parameters. The procedure is as follows. First, AOP4CSM is based on four specific join points. Each one corresponds to an instant (t_1, t_2, t_3 and t_4) where AOP4CSM intercepts method calls. Second, AOP4CSM executes the corresponding advice (for each join point) which consists of (1) saving the timestamp and (2) calculating the number of invocations at the client (advices 1 and 4). When all instants have been processed, AOP4CSM computes the difference between t_4 and t_1 to deduce the response time. It also subtracts t_2 from t_3 to calculate the execution time. The difference between the response time and the execution time represents the communication time.

AOP4CSM assesses the throughput by (1) calculating the number of successful invocations at the client side (fourth join point) [SuccRequests] and (2) dividing this number by a period of time [T] (in our case T is equal to 10 minutes) as

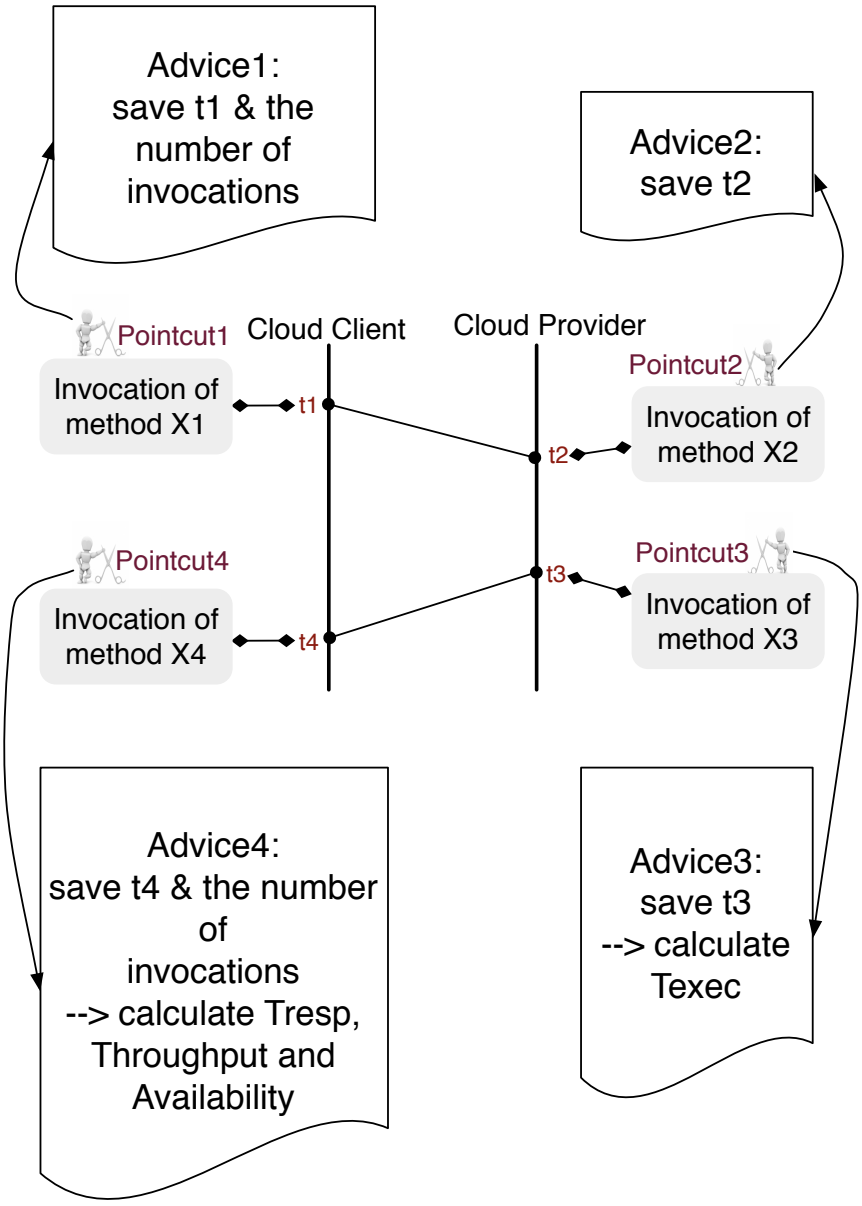

Figure 4.3: Functionality of AOP4CSM

described in Equation (4.2). Moreover, AOP4CSM calculates the availability as described in Equation (4.3).

AOP4CSM fulfills the first requirement of this thesis **R1**, since it neither modifies the server implementation, nor the client code.

The multi-layer monitoring agent operates on all Cloud layers and collects related parameters. Therefore, it fulfills the second requirement of this thesis **R2**.

PI_Monitor, VI_Monitor, P_Monitor and S_Monitor make use of basic TCP/IP sockets to send collected data to the Elementary Event Collector (EVC). The EVC sends monitored data to the analysis agent of CEP4Cloud. It is detailed in Section 4.4.

4.4 Analysis

An important component of CEP4Cloud is the analysis agent. It is based on the methodology of CEP and is used to detect performance-related problems, while identifying their causes and the corresponding layers (physical infrastructure, virtual infrastructure, platform and software). The analysis agent is based on two new analysis approaches. The first one is called CEP4CMA for "**C**omplex **E**vent **P**rocessing for **C**loud **M**onitoring and **A**nalysis". The novelty of CEP4CMA is that the CEP queries (i.e., analysis rules) are derived from a comprehensive analysis of the relationships between monitored metrics across Cloud layers. CEP4CMA is presented in Section 4.4.1. The second analysis approach mainly deals with the architectural design of the analysis agent and is called D-CEP4CMA for "**D**ynamic **C**omplex **E**vent **P**rocessing for **C**loud **M**onitoring and **A**nalysis". The basic idea of D-CEP4CMA is to dynamically switch between different CEP architectures depending on the current conditions of the observed Cloud environment. D-CEP4CMA is described in Section 4.4.2.

4.4.1 CEP4CMA

Figure 4.4 depicts the architecture of CEP4CMA. It makes use of a set of cross-layer analysis rules (i.e., queries) implemented within a CEP engine. In this work, the Esper CEP engine has been used. Thus our analysis rules are implemented as EPL queries.

The used analysis rules are defined on the basis of relationships between the monitored parameters across Cloud layers.

In particular, extracting relationships (i.e., correlations) between metrics has two main benefits. First, it allows us to reduce the number of monitored parameters, since analyzing two "highly" correlated metrics gives the same result as analyzing one of these two metrics. Second, the relationships are very useful to define the analysis rules. Consequently, it is possible to rapidly detect a performance degradation (thanks to the reduced number of metrics) and accurately identify its layer and its cause (thanks to the intelligent analysis rules).

To identify the relationships between the monitored parameters, we have used statistical methods, such as the calculation of the correlation and multiple correlation coefficients. Thus, we performed several experiments that consist of (1) monitoring the defined parameters (see Table 3.1); and (2) computing, via different statistical methods, the correlation between them. The conducted ex-

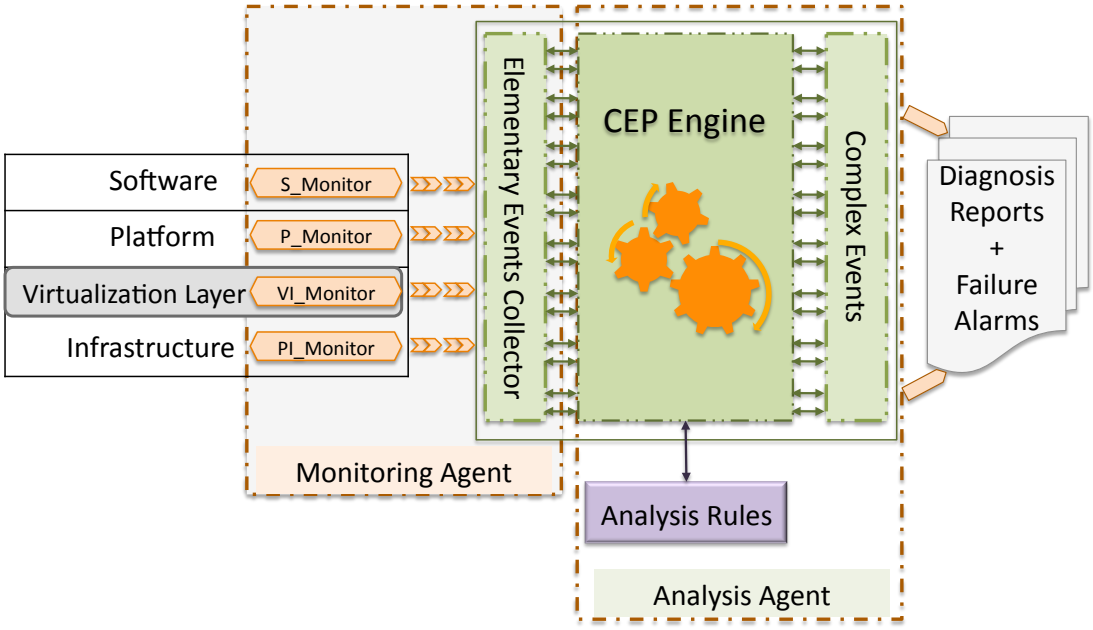

Figure 4.4: The architecture of CEP4CMA

periments, results and obtained conclusions showing the relationships between parameters across Cloud layers are detailed in the next section and summarized in Table 4.4. Our experiments have been performed on serveral samples. The sample size depends on the scenario. It varies between 20 and 100 data points. A data point represents one measurement of the studied metric. Table 4.4 shows the number of data points (Pnb) for all experiments. It should be pointed out that Table 4.4 does not state the correlation coefficient and the number of data points of straightforward relationships. For instance, the correlation coefficient between the free disk and the used disk is not given.

Relationships between Cloud Layers

To study the relationships between Cloud layers, we followed two steps. First, the interactions between metrics across Cloud layers have been theoretically examined. Second, several experiments have been conducted to verify the theoretically obtained relationships. The experimental results are used to remove incorrect theoretical relationships, keep the correct ones and update the not well defined ones. This thesis only presents the verified relationships (theoretically defined and experimentally verified).

The conducted experiments allow us to compute statistical indicators like the correlation and the multiple correlation coefficients, in order to prove the identified relationships. They are partitioned into three groups. The first group of experiments consists of measuring two metrics in normal conditions (without

generating any load) and calculating the correlation coefficient. According to its value, we deduce the relationship between the two metrics [94]. The second group of experiments deals with the case when there are more than two related metrics. The related metrics are monitored in normal conditions and the multiple correlation coefficient between them is computed [23]. The G*Power [31, 38] tool is used to compute the multiple correlation coefficient. The last group of experiments consists of generating load with respect to the first parameter and observing its effects on the second one. If both increase, decrease or inversely vary together, this means that they are related. Since in this step we deal with correlation experiments, we use a small testbed. It allows us to isolate the studied metrics and easily identify their relationships. The testbed for the correlation experiments is composed of one physical node with 1 GB of RAM and 100 GB of disk, running under the Debian operating system. Xen 4.1 was chosen as the virtualization technology. It allows us to administer virtual machines through its hypervisor and its privileged domain Dom0 [42]. Moreover, it manages access to the hardware resources, such as the disk and the memory, via Dom0 and the hypervisor. The testbed for the correlation experiments has a Cloud architecture with four layers. The hardware resources, Xen, its hypervisor and Dom0 constitute the physical infrastructure layer. The virtual infrastructure layer is composed of the DomU virtual machines. The used platform layer consists of the Java Virtual Machine (JVM) with Apache Tomcat as a web server. Under Tomcat, the Axis engine is deployed to manage web services. The Axis engine and the web services constitute the software layer (see Figure 4.5).

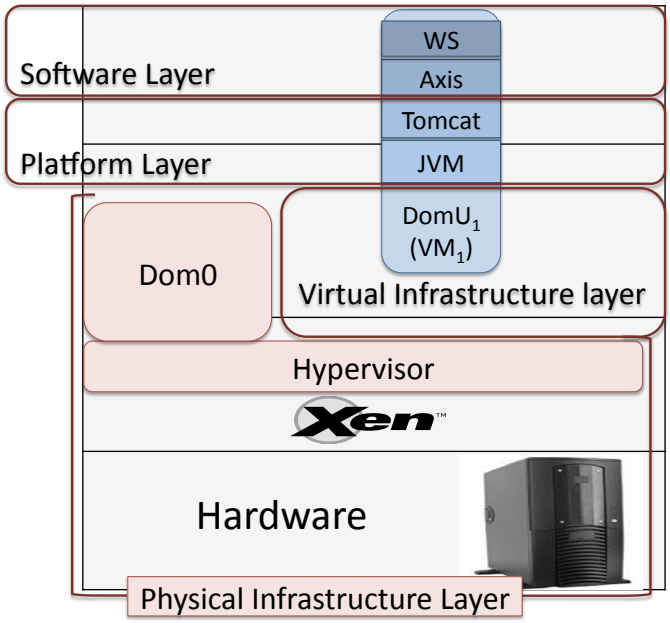

Figure 4.5: Testbed for correlation experiments

To extract relationships between the monitored metrics in a theoretical study, we classified the monitored metrics according to two criteria: the layer of a metric (e.g., the infrastructure layer) and the category of a metric (e.g., the CPU category). Thus, we have four groups of relationships:

- **Intra-Category, Intra-Layer Relationships** describe relationships between metrics belonging to the same layer and the same category. The majority of these relationships are used to reduce the number of parameters. For example, experiments confirmed that the CPU time of a running thread is related to its *waited count* by calculating the correlation coefficient (0.9) of the two parameters for 50 data points. Such a relationship allows us to reduce the number of monitored parameters. It is sufficient to monitor only one of the two metrics, since they describe the same information.

- **Intra-Category, Inter-Layer Relationships** describe relationships between metrics belonging to the same category. They are used to define the analysis rules. For example, an experiment based on 30 data points has shown that the CPU time of a Java thread is related to the *CPU user time* of its virtual machine, since the corresponding correlation coefficient is equal to 0.5. This observation allows us to deduce the cause of a VM CPU performance-related problem.

- **Inter-Category, Intra-Layer Relationships** describe relationships between metrics belonging to the same Cloud layer. They are useful to reduce the number of observed metrics if this reduction does not affect the quality of the analysis. Otherwise, these relationships are used to define the analysis rules. For example, an experiment based on 65 data points has shown that the number of running processes is related to the machine load, since the correlation coefficient is equal to 0.9. Such a relationship is used to define the analysis rules.

- **Inter-Category, Inter-Layer Relationships** describe relationships between metrics belonging to different layers and different categories, such as the relationship between the I/O requests to the physical disk and the number of Bytes In and the number of Bytes Out. This relationship is illustrated by calculating the multiple correlation coefficient (0.8) of the I/O disk requests and Bytes In and Bytes Out for 30 data points. The majority of such relationships are used in the analysis rules.

Table 4.4 shows the classification of the metrics and the extracted relationships (i.e., correlations). In the columns, we have the four Cloud layers: the physical infrastructure, the virtual infrastructure, the platform and the software layers. In the rows, we have the different categories: CPU, memory, disk, load, network, interrupts, Java classes and time related quality-of-service parameters.

We start by studying simple relationships inside a category and a layer, such as the relationship between the CPU metrics at the virtual infrastructure layer. The first extracted relationship can be used to reduce the number of monitored CPU metrics by removing the CPU idle time. Actually, the CPU idle time depends on other CPU metrics (user, system, wait and steal). Thus, it is not necessary to monitor the CPU idle time. Figure 4.6 shows the conducted experiment demonstrating this relationship.

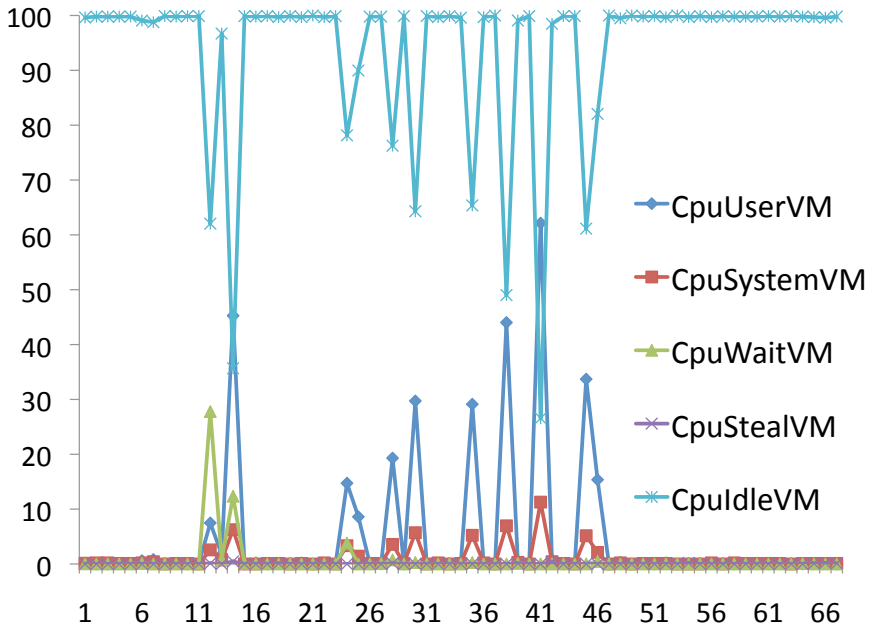

Figure 4.6: Relationship between CPU metrics and CPU idle time

Moreover, experiments (see Figure 4.7) showed that the CPU time of a running thread is highly related to its waited count. This means that it is sufficient to monitor one of these two metrics. We have chosen to monitor the CPU time.

In the group of inter-layer relationships, we found that the thread CPU time is related to the CPU user of its virtual machine (see Figure 4.8). Due to the high number of threads running on a virtual machine, only two threads are presented in Figure 4.8. The two correlation coefficients are 0.4 and 0.5, respectively. Thus, the CPU user of a virtual machine is related to the CPU time of the running threads.

Moreover, the relationship between the CPU user metrics at the virtual and physical (hypervisor) layers has been studied. Figure 4.9 shows the correlation between the Dom0 CPU usage and the virtual machine CPU user. In this scenario, we have one virtual machine running on a physical node that contains two cores. Figure 4.9 shows that the usage of the physical core 1 is highly related to the CPU user of the virtual machine since the correlation coefficient value is equal to 0.8. However, Figure 4.9 also shows that the core 0 usage is not related

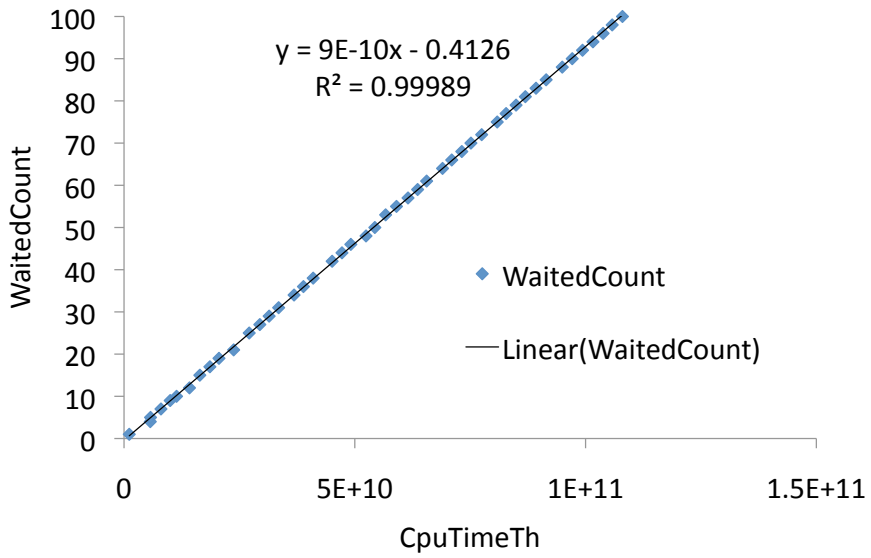

Figure 4.7: The CPU time of a thread is highly correlated to its waited count; the correlation coefficient is equal to 0.99.

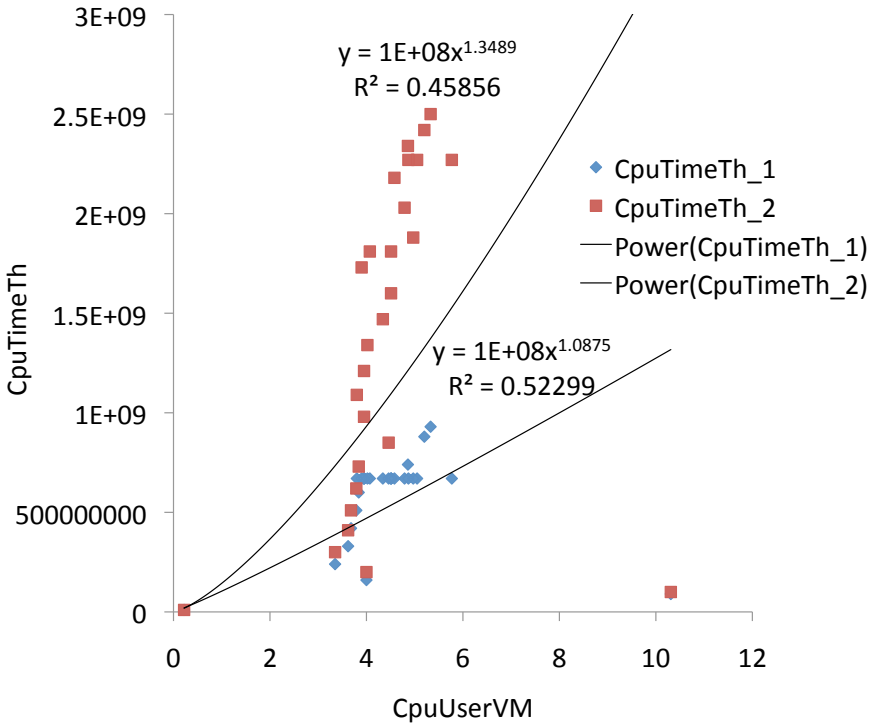

Figure 4.8: The CPU time of a thread is related to the CPU user of the corresponding virtual machine.

to the VM CPU user since the correlation coefficient value is around 0. This is

actually related to the fact that the virtual machine is only using core 1 in this scenario. The multiple correlation coefficient between the virtual machine CPU user and the CPU usage of the two cores is around 0.8, which implies a strong relationship between these metrics.

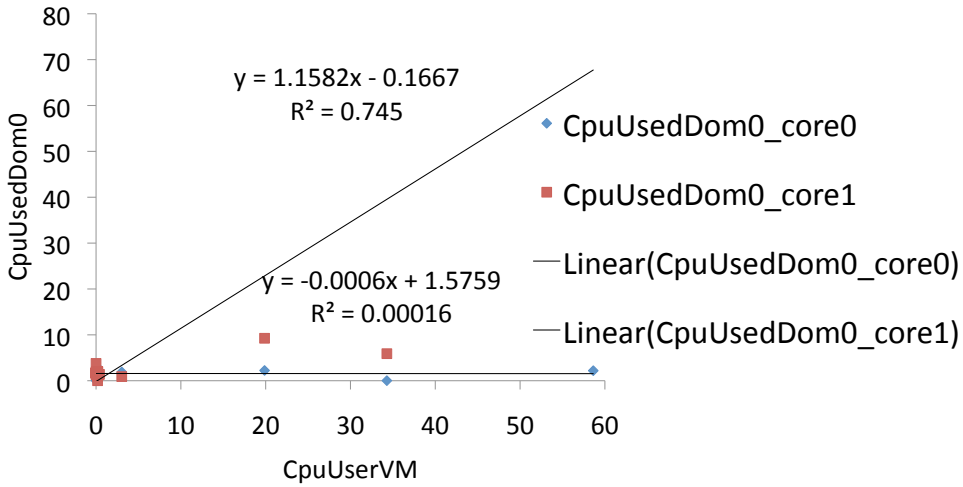

Figure 4.9: The CPU user of the virtual machine is related to the Dom0 CPU usage.

In addition, we verified that the VM CPU steal is related to the DomU waiting time. Figure 4.10 shows the relationship between the virtual machine CPU Steal and its waited time to access the physical CPU. In this scenario, we generated CPU load on the virtual machine (via a script calculating the factorial of a big number), and we observed its effect on the waited time on core 0 and core 1 of the CPU. This experiment shows that the considered parameters increase together. The correlation coefficients of the VM CPU steal and the VM CPU waiting time of Core 0 and Core 1 are 0.45 and 0.40, respectively. This implies that they are related.

In the memory category, we observed that the JVM heap Memory is related to the free memory of the VM. As shown in Figure 4.11, 3 JVMs are running on the used testbed. They correspond to the Tomcat web server, the used web service and the JVMSensor tool. The conducted experiments show that the heap memory of JVM1 and JVM3 are correlated to the free memory of the VM. However, they also show that the JVM2 heap memory is not correlated with the VM free memory. This does not necessarily invalidate the relationship between the JVM heap memory and the VM free memory. Actually, we are studying the correlation between one parameter (VM free memory) and three other parameters (JVM1 / JVM2 / JVM3 heap memory). In such a case, it is more significant to calculate the multiple correlation coefficient. It is equal to 0.81, which reflects a strong relationship between the JVM heap memory and the VM free memory.

Like the JVM heap memory, the non-heap memory is also related to the

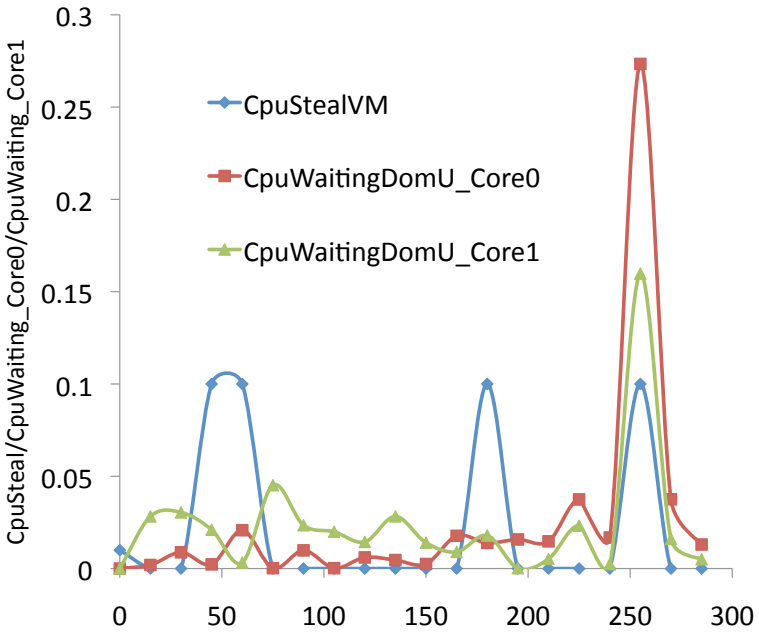

Figure 4.10: Relationship between the CPU steal of the virtual machine and its waited time to access the CPU

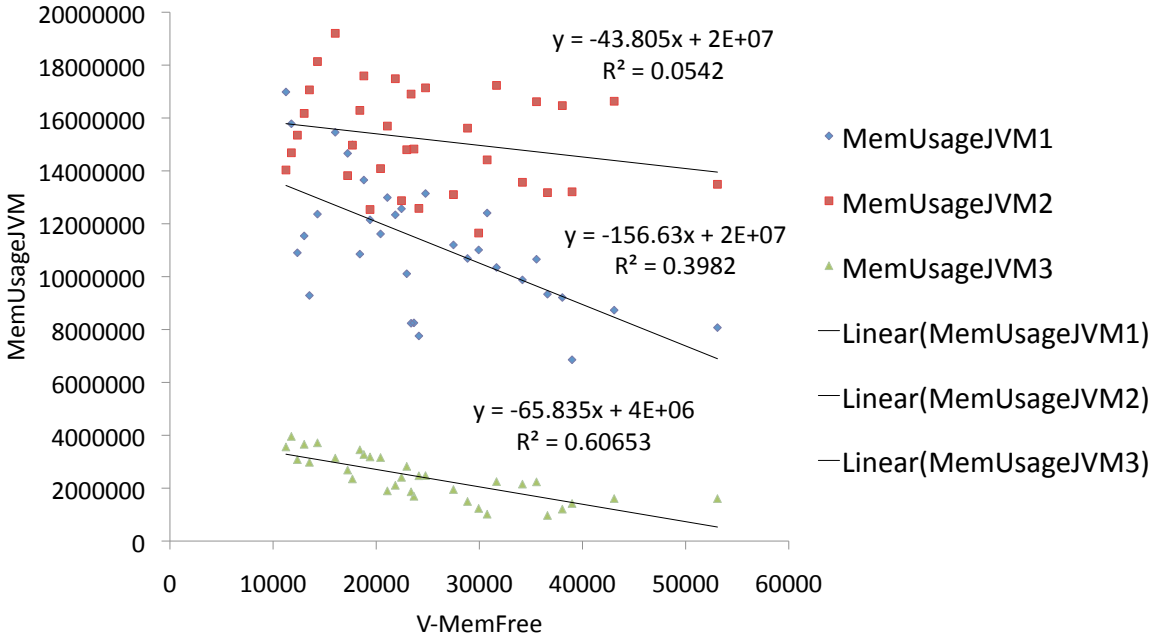

Figure 4.11: The JVM memory usage is negatively correlated with the free memory of the virtual machine.

virtual machine free memory. Figure 4.12 depicts the results of the conducted experiments. We are still using the same scenario with 3 JVMs. It is visually

clear that the non-heap memory values of JVM1 and JVM3 are highly correlated to the VM free memory (see Figure 4.12). Morevover, it is easy to notice that the

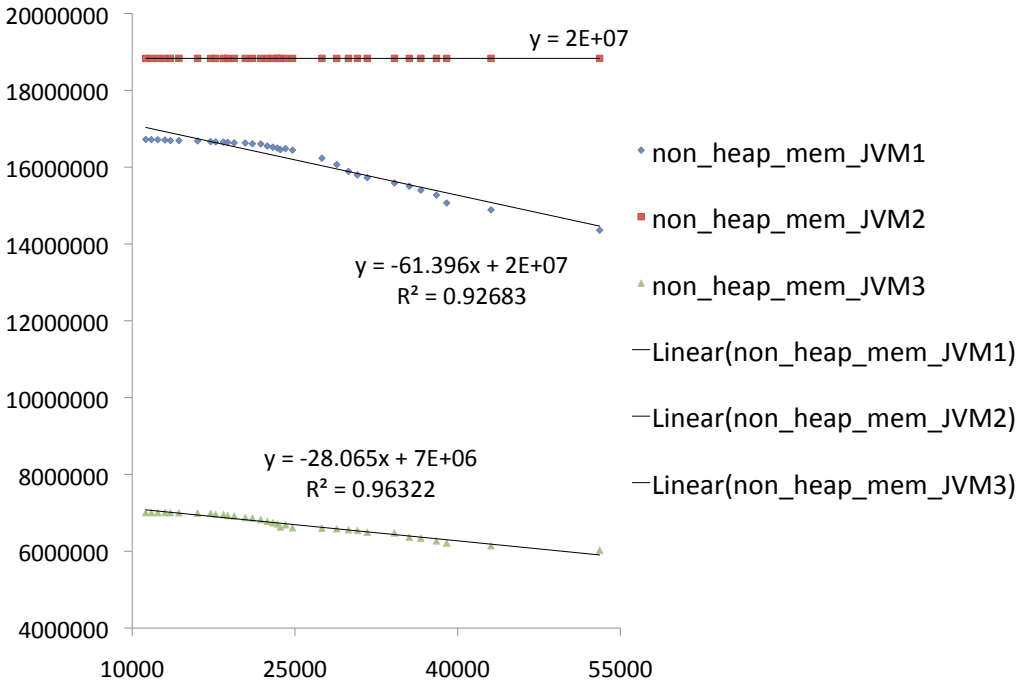

Figure 4.12: The non-heap JVM memory is negatively correlated with the free memory of the virtual machine.

JVM2 heap memory distribution is constant. Thus, it is not possible to calculate the correlation coefficient in this case. Therefore, it is not correlated with the VM free memory. Nevertheless, it is still possible to confirm a relationship between the JVM non-heap memory and the VM free memory. Actually, it is not surprising, in a multiple correlation scenario, that one of the parameters (JVM2 non heap memory) is not correlated with the main parameter (VM free memory), when the two other parameters (JVM1 / JVM3 non-heap memory) are highly correlated to the same parameter (VM free memory).

Table 4.4: Cloud parameters: classification and relationships

	Physical Layer	Virtual Layer	Platform Layer	Software Layer
CPU Metrics	CpuUsedDom0 CpuBlockedDomU CpuWaitingDomU ExecSec	CpuUserVM CpuWaitVM CpuSystemVM "*CpuIdleVM*" CpuStealVM	CpuTimeTh "*WaitedCount*"	
Intra-category, Intra-layer Relationships		CpuUserVM = -f(CpuIdleVM)[1] CpuWaitVM = -f(CpuIdleVM) CpuSystemVM = -f(CpuIdleVM) CpuStealVM = -f(CpuIdleVM) Pnb=70, See Figure 4.6	CpuTimeTh = f(WaitedCount)[2] See Figure 4.7 Corr(CpuTimeTh, WaitedCount)=0.9 Corr is the correlation coefficient Pnb=50	
Intra-category, Inter-layer Relationships	CpuTimeTh=f(CpuUserVM), Pnb=30, see Figure 4.8 CpuUserVM=f(CpuUsedDom0), MCC(CpuUserVM, CpuUsedDom0Core)=0.8, Pnb=20, see Figure 4.9 CpuStealVM=f(CpuWaitingDomU), Corr(CpuStealVM, CpuWaitingDomU)=0.4, Pnb=20, see Figure 4.10 MCC is the multiple correlation coefficient			
Memory Metrics	P-RamFree "*P-RamUsed*" "*P-SwapIn*" "*P-SwapOut*" P-SwapFree P-Read P-Wrtn	V-RamFree "*V-RamUsed*" "*V-SwapIn*" "*V-SwapOut*" V-SwapFree V-Read V-Wrtn	HeapMem NonHeapMem	
Continued ...				

[1]-f() indicates a negative relationship between the two parameters: If the first parameter increases, the second one decreases and vice versa.
[2]f() indicates a positive relationship between the two parameters: If the first parameter increases, the second one increases and vice versa.

... Continued	Physical Layer	Virtual Layer	Platform Layer	Software Layer
Intra-category Intra-layer Relationships	P/V-SwapIn = -f(P/V-SwapFree) P/V-SwapOut = -f(P/V-SwapFree) P/V-RamUsed = -f(P/V-RamFree) P/V-RamFree = 0 "implies" P/V-SwapFree decreases			
Intra-category Inter-layer Relationships	HeapMem = -f(V-RamFree), MCC(V-RamFree, HeapMem)=0.8, Pnb=35, see Figure 4.11 NonHeapMem = -f(V-RamFree), MCC(V-RamFree, NonHeapMem)=0.9, Pnb=35, see Figure 4.12 V-RamFree = f(P-RamFree), Corr(V-RamFree, P-RamFree)=0.9, Pnb=60, see Figure 4.13			
Disk Metrics	P-DiskFree *"P-DiskUsed"* IOReqDisk	V-DiskFree *"V-DiskUsed"*		
Intra-category, Intra-layer Relation-ships	P/V-DiskUsed = -f(P/V-DiskFree)			
Inter-category, Intra-layer Relationships	P/V-SwapFree = f(P/V-DiskFree) P/V-Read = -f(P/V-DiskFree), P/V-Wrtn = -f(P/V-DiskFree) MCC(P/V-DiskFree, (P/V-Wrtn, P/V-Read))=0.83 IOReqDisk = f(P-Read, P-Wrtn), Pnb=100, see Figure 4.14			
Continued ...				

	Physical Layer	Virtual Layer	Platform Layer	Software Layer
... Continued				
Intra-category, Inter-layer Relation-ships	P-DiskFree = f(V-DiskFree), Corr(P-Disk, V-Disk) = 0.4, Pnb=100			
Inter-category, Inter-layer Relationships	IOReqDisk = -f(V-RamFree), Corr(IOReqDisk, V-RamFree)=-0.8, Pnb=30 IOReqDisk = f(CpuBlockedDomU), Corr(IOReqDisk, CpuBlockedDomU)=0.3, Pnb=30 IOReqDisk = -f(ExecSec), Corr(IOReqDisk, ExecSec)=-0.1, Pnb=30 IOReqDisk = f(CpuWaitingDomU), Corr(IOReqDisk, CpuWaitingDomU)=0.2, Pnb=30 The square of the multiple correlation coefficient=0.2, Pnb=30 Thus, the multiple correlation coefficient=0.4[3]			
Load	P-Load	V-Load		
Intra-category, Inter-layer Relation-ships	V-Load = f(P-Load), Corr(V-Load, P-Load)=0.77, Pnb=65, see Figure 4.15			
Inter-category, Inter-layer Relationships	IOReqDisk = f(P-Load), Corr(IOReqDisk, P-Load) is about 0.4, Pnb=30			
Network	P-BytesIn P-BytesOut	V-BytesIn V-BytesOut		
Continued ...				

[3]The multiple correlation coefficient is computed via the G*Power tool. It takes as inputs (1) the correlation coefficient values between the outcome (IOReqDisk) and their predictors (CpuBlockedDomU, ExecSec, CpuWaitingDomU) and (2) the correlation values between predictors.

	Physical Layer	Virtual Layer	Platform Layer	Software Layer
... Continued				
Inter-category, Inter-layer Relationships	IOReqDisk = -f(V-BytesIn), Corr(IOReqDisk, V-BytesIn)=-0.82, Pnb=30 IOReqDisk = -f(V-BytesOut), Corr(IOReqDisk, V-BytesOut)=-0.34, Pnb=30 The square of the multiple correlation coefficient=0.72 Thus, the multiple correlation coefficient=0.8			
Processes	P-ProcNb	V-ProcNb		
Inter-category, Intra-layer Relationships	P-ProcNb = f(P-Load) Corr = 0.9, Pnb=65 See Figure 4.17	V-ProcNb = f(V-Load) Corr = 0.9, Pnb=65 See Figure 4.17		
Interrupts	HwInterr	SwInterr	ExcepNb	
Inter-category, Inter-layer Relationships	CpuWaitVM = f(HwInterr) CpuWaitVM = f(SwInterr), Corr(CpuWaitVM, SwInterr) = 0.8, Pnb=100, see Figure 4.18 CpuWaitVM = f(ExcepNb)			
Classes			"*ClassNb*"	
Inter-category, Intra-layer Relationships			ClassNb = f(HeapMem), Corr(ClassNb, HeapMem)=0.9, Pnb=35 see Figure 4.16	
Time Related QoS parameters				Texec Tcom "*Tresp*"
Continued ...				

... Continued	Physical Layer	Virtual Layer	Platform Layer	Software Layer
Intra-category, Intra-layer Relation-ships				Tresp = Texec + Tcom [69]
Trivial Symptoms of a perfor-mance degradation	A **High Disk Consumption** leads to the degradation of **Texec** A **High CPU Consumption** leads to the degradation of **Texec** A **High Memory Consumption** leads to the degradation of **Texec** A **High Load** leads to the degradation of **Texec** A **High Network Usage** leads to the degradation of **Tcom**			
End				

Figure 4.13 shows that the VM free memory is highly related to the free memory of the physical machine. The correlation coefficient is around 0.9.

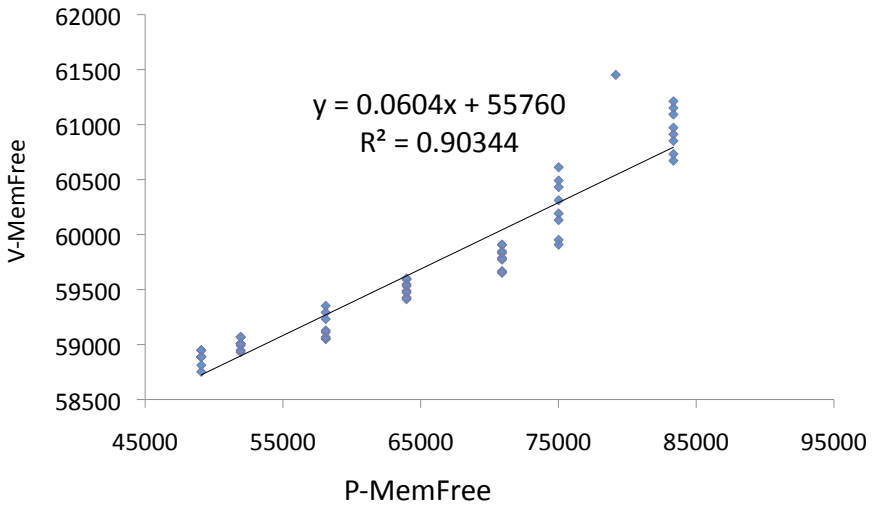

Figure 4.13: Relationship between the physical machine free memory and the VM free memory

It is plausible that the number of read/written pages is related to the number of I/O requests to the disk. This relationship has been proven via experiments. As shown in Figure 4.14, the number of written pages is correlated to the number of I/O requests to the disk.

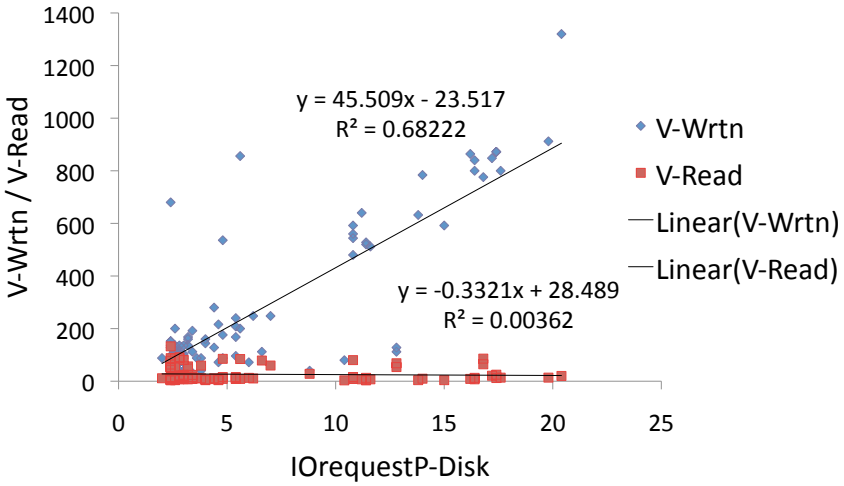

Figure 4.14: Relationship between the I/O requests to the physical disk and the number written/read pages of the virtual machine

However, the number of read pages is not, since it almost follows a constant distribution. The multiple correlation coefficient between the number of I/O

requests to the disk and the number of read/written pages is equal to 0.83. This value implies a strong relationship between the corresponding parameters.

In addition, it has been proved that the privileged domain (Dom0) load is related to the VM load. Figure 4.15 shows that the correlation coefficient between the two loads (Dom0 and VM) is about 0.77, which implies a high correlation/relationship between the virtual load and the physical one.

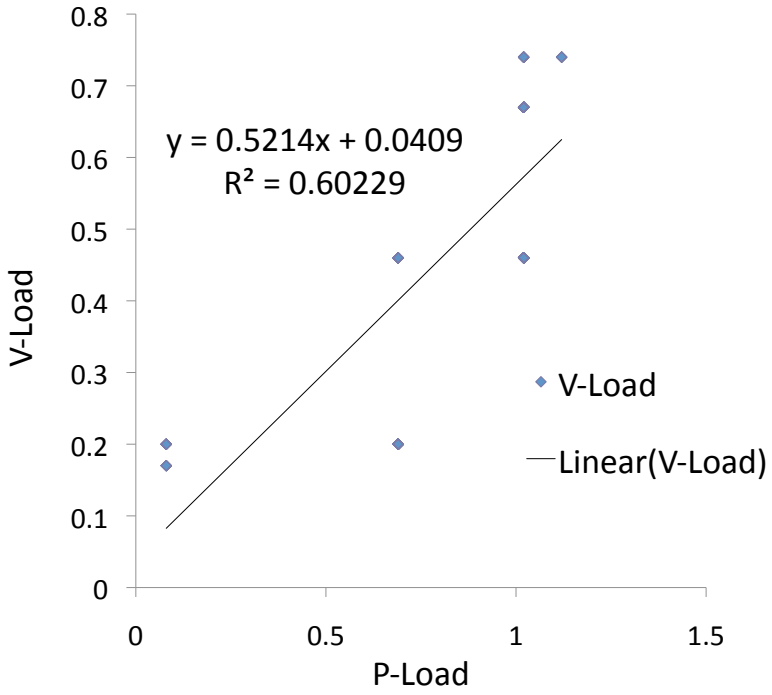

Figure 4.15: The load of the virtual machine is highly related to the load of the privileged domain (Dom0).

In the platform layer, a strong relationship between the number of loaded classes and the Non-Heap memory usage has been proved (see Figure 4.16). Actually, the number of loaded classes is highly correlated to the non-heap memory usage. The correlation coefficient is about 0.91. This means that the increase of the non-heap memory usage is mainly related to an increase of the number of loaded classes. Thus, it is not necessary to measure the number of loaded classes.

The relationship between the number of processes and the machine load has also been studied. This study demonstrates that the machine load is related to the number of processes. Figure 4.17 shows the corresponding experimental result.

Moreover, there are relationships between the interrupts and the virtual machine CPU wait. Interrupts create context switches that lead to the increase of the CPU wait. This means that interrupts (hardware interrupts, software interrupts and exceptions) are related to the virtual machine CPU wait. Figure

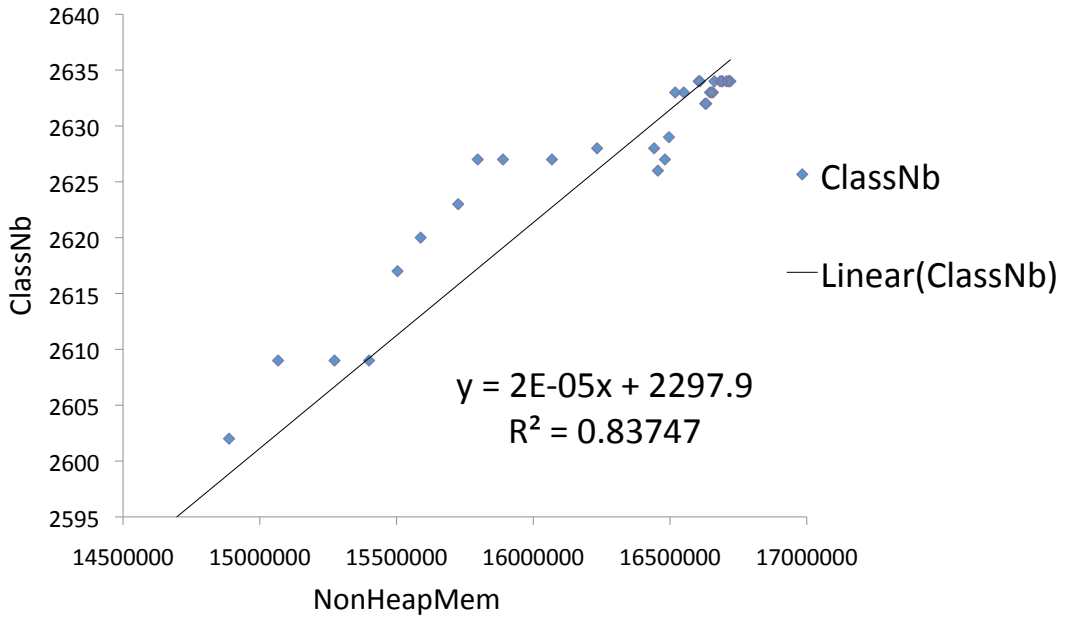

Figure 4.16: Correlation between the number of loaded classes and the non-heap memory usage

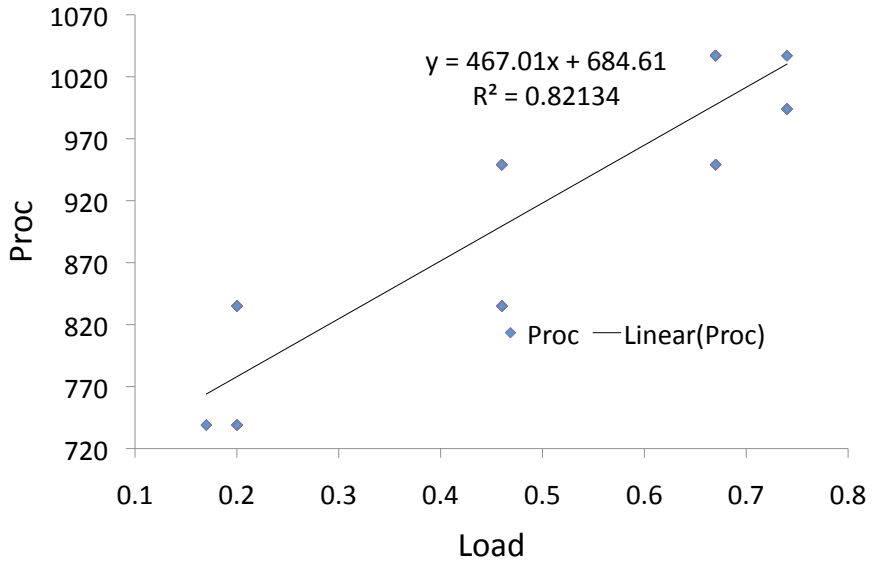

Figure 4.17: Relationship between the number of processes and the load of a machine

4.18 shows the correlation between the number of the Dom0 software interrupts and the virtual machine CPU wait. The majority of points follows the same

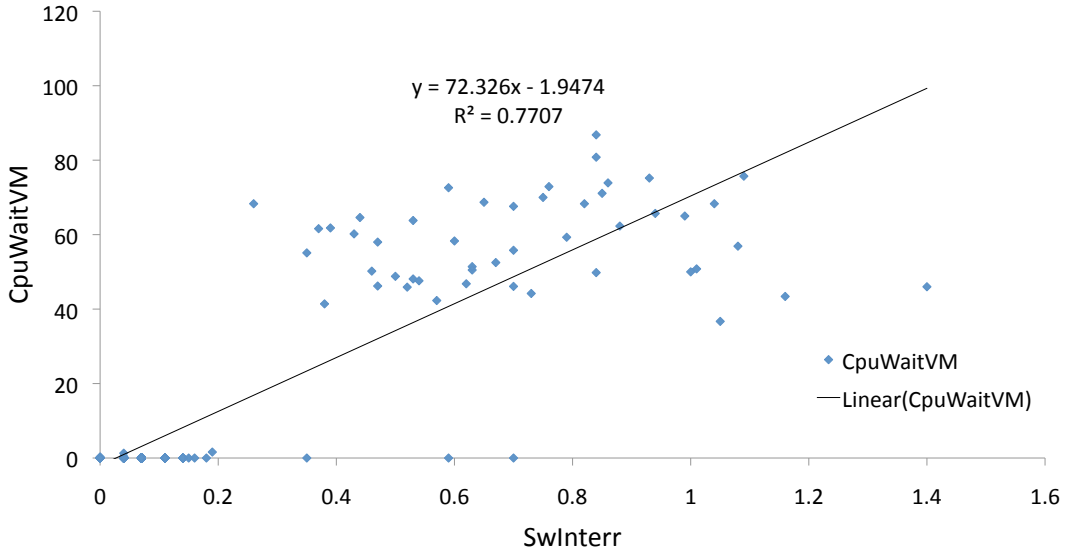

Figure 4.18: Software interrupts are highly related to the CPU wait of a virtual machine in the absence of exceptions and hardware interrupts

linear spectrum. The correlation coefficient is about 0.877, which indicates a relationship between the VM CPU wait and the software interrupts number of Dom0.

In the software layer, the time related QoS parameters are also depending on each other. Actually, the response time is equal to the sum of the execution and the communication time [69]. Thus, measuring the response time will not bring new information to the analysis. This is the reason why we only monitor the execution and the communication times.

It should be pointed out that the correlation coefficient reflects the relationship between the corresponding parameters. A positive / negative correlation is deduced when the correlation coefficient is near to 1 / -1. Values in between, where the absolute values vary from 0.3 to 1, reflect a relation between the treated metrics. Crocker [23] has shown that a correlation coefficient equal to 0.3 could reflect a strong causal link, in case the relationship is meaningful. He presents the example of the causal relationship between smoking and lung cancer, and he assumes that a correlation coefficient equal to 0.3 is very significant in this case. We believe that Crocker's observations are also valid in our scenario. For instance, it is obvious that the CPU idle percentage is related to all other CPU percentages (such as CPU user, CPU system and CPU wait). Such correlations can be directly deduced from the definition of the metrics. According to our observations, the correlation coefficient of the CPU idle and CPU wait is around 0.4. This value implies a correlation between the two metrics, since the rela-

tionship is meaningful. This is what we do in this study: presenting meaningful relationships between Cloud metrics (deduced by thinking) and showing these relationships via experiments. In the experimental step, a correlation coefficient value very close to 1/-1 is not required to deduce a relationship (0.4 could be significant and sufficient).

As mentioned earlier, extracting relationships between parameters across Cloud layers allows us to reduce the number of gathered metrics. All removed parameters are written in italics (see Table 4.4). Moreover, these relationships are used to define the analysis rules. The next section details the proposed analysis rules, the associated cause-effect (Fishbone) diagrams and their corresponding analysis rules.

The Analysis Rules

The definition of analysis rules is based on the extracted relationships between metrics across Cloud layers (see Table 4.4), while adopting a **R**oot **C**ause **A**nalysis (RCA) approach.

We use fishbone diagrams to perform Cloud performance analysis. Thus, we start the analysis by stating the trivial causes of a performance-related problem (see Table 4.4). Starting the analysis by studying such evident causes is necessary, but not sufficient to give accurate information about the nature of a degradation. Figure 4.19 shows a high-level view of our cause-effect diagram.

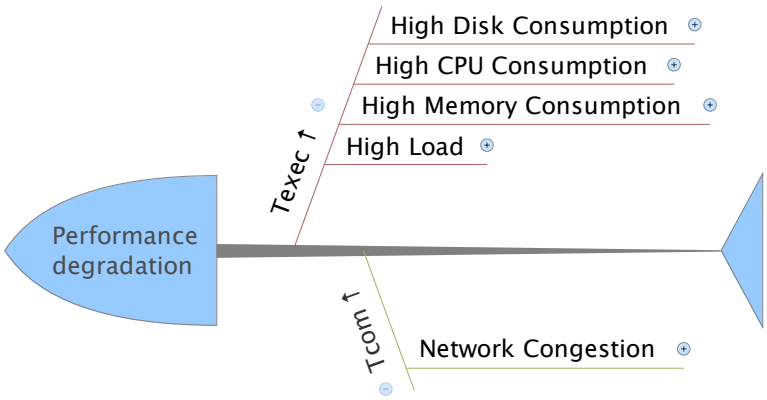

Figure 4.19: The cause-effect diagram: A high level view

As shown in Figure 4.19, the first step of the analysis only states trivial causes of a performance degradation. Stopping the analysis in this step could lead, in some cases, to wrong results, due to the interaction between Cloud metrics. This is the reason why it is necessary to continue the analysis of these evident causes. For instance, this first analysis (see Figure 4.19) shows that a communication time degradation is always related to a network congestion (low throughput). However, expanding this branch on the basis of the extracted relationships demonstrates

that such a degradation could also be related to a huge number of requests to the physical disk (see Figure 4.20).

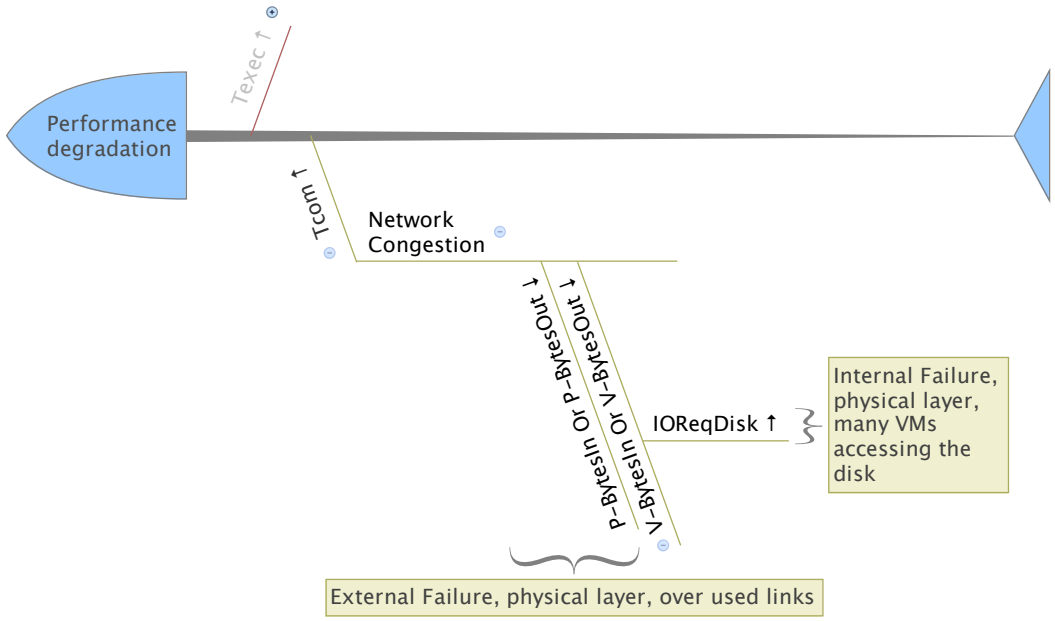

Figure 4.20: The cause-effect Diagram: The analysis of a communication time degradation

Next, we expand branches related to the execution time analysis. The first branch deals with the case when an execution time degradation is related to a high load. According to the extracted relationships, we conclude that this high load could be related to a large number of processes or to a continuous increase of the number of I/O requests to the physical disk. As presented in Figure 4.21, diagnosis reports accurately show the cause of the performance degradation.

The next branch deals with the case when an execution time degradation is related to a high CPU consumption (see Figure 4.22). A simple analysis shows that the cause is certainly related to the CPU. However, this is not true in most of the cases. Actually, the extracted relationships show that such degradation could be related to disk problems or to a large number of interrupts. Figure 4.22 shows this branch. It details the origins of a CPU exhaustion. As shown in Figure 4.22, a high CPU consumption is related to six different causes:

- An increase of the Dom0 CPU user that is related, according to the extracted relationships, to an increase of the VM CPU user. The latter is caused by an increase of threads' CPU time. This means that a high CPU consumption is related, in this case, to the large number of threads due to the fact that the waited count of a thread is highly correlated to its CPU time.

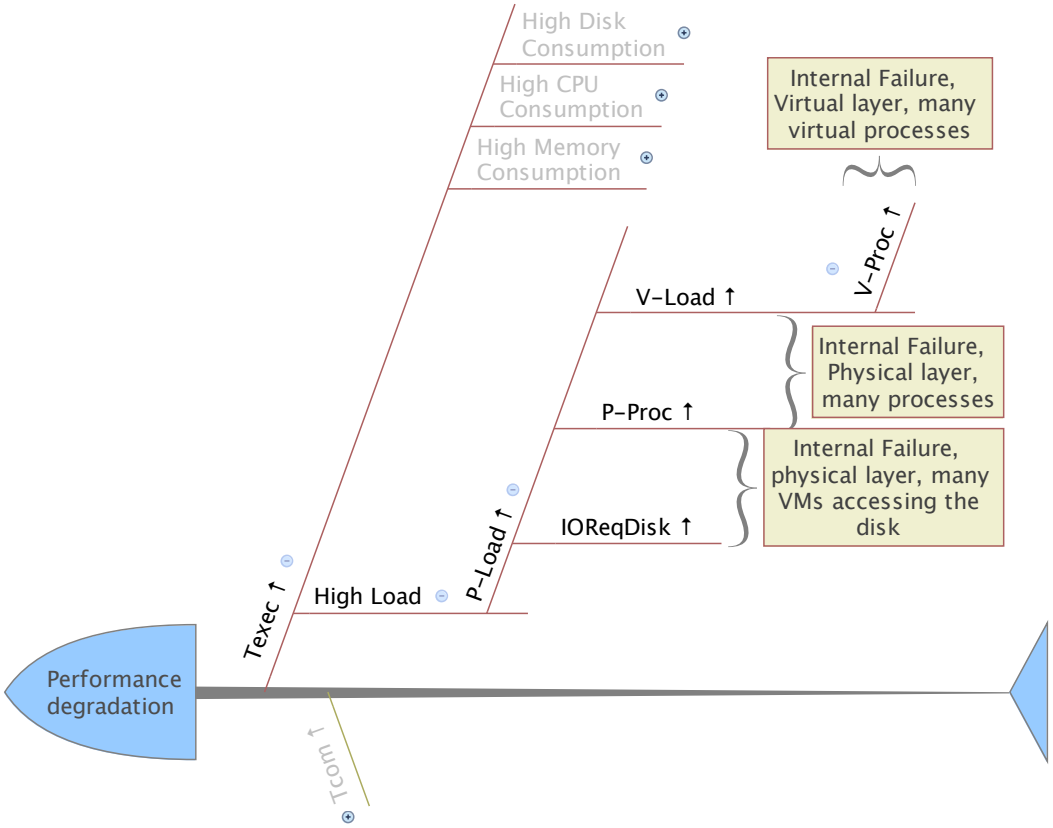

Figure 4.21: The analysis of an execution time degradation: The load branch

- A degradation of the blocked time of DomU (running virtual machines) that is mainly related to an increase of the number of I/O requests to the physical disk.
- An increase of the waiting time of DomU that could be related to an increase of the number of I/O requests to the physical disk or to an increase of the CPU Steal.
- A decrease of the number of the executions per second that is also caused by an increase of the number of I/O requests to the physical disk.
- An increase of the virtual machine CPU wait that could be related to a large number of interrupts (hardware interrupts, software interrupts and exceptions).
- An increase of the virtual machine CPU system, which means that the virtual machine is overloaded and needs more virtual resources.

Figure 4.22 shows the deduced diagnosis reports. They point out the fact that a CPU problem could be related to another resource (different to CPU).

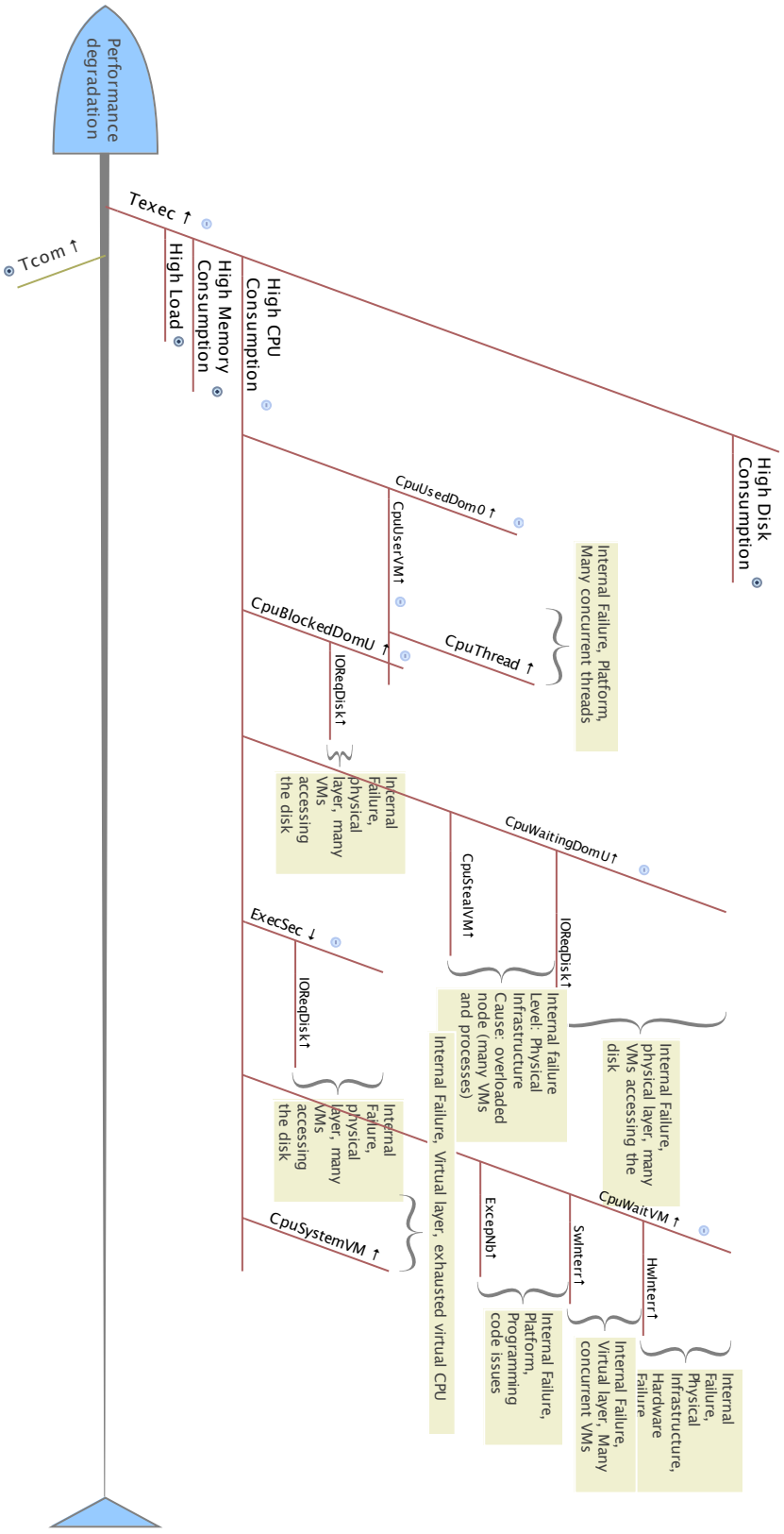

Figure 4.22: The analysis of an execution time degradation: The CPU branch

The following branch of the cause-effect diagram (see Figure 4.23) deals with the case when the degradation of the execution time is caused by a high memory consumption. Since the virtual machine free memory is highly related to the physical machine free memory, we deduce that a high memory consumption is mainly caused by a decrease of the VM free memory. The latter, as demonstrated in Table 4.4, could have three different origins. It could be related to an increase of the JVM heap memory. In this case, we deduce that the origin of the performance problem is the huge number of Java objects. Thus, launching the garbage collector could solve the problem. A decrease of the VM free memory could be also related to an increase of the JVM non-heap memory. In this case, we deduce that the cause of the performance degradation is the huge number of loaded classes since the number of loaded classes is highly correlated with the non-heap memory usage. Moreover, the exhaustion of the VM free memory could be related to the large number of I/O requests to the physical disk.

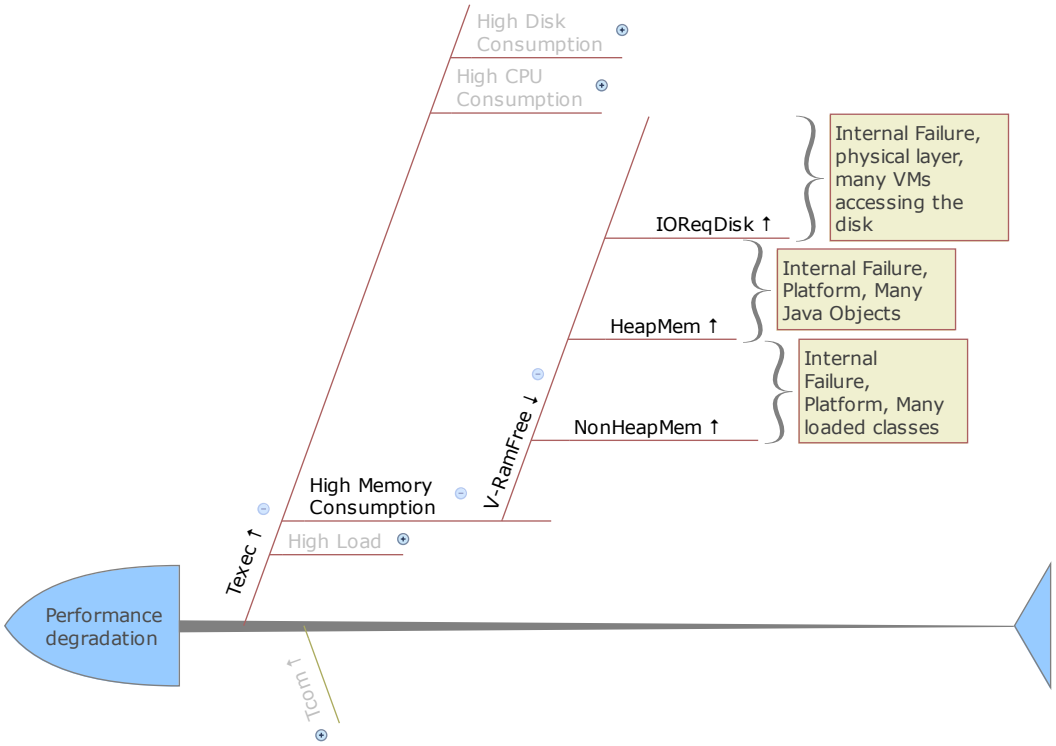

Figure 4.23: The analysis of an execution time degradation: The memory branch

The last branch of the cause-effect diagram deals with the case when an execution time degradation is caused by a high disk usage. It is presented in Figure 4.24 and shows the possible causes of a high disk consumption. As already discussed, the free physical disk is related to the free VM disk. Thus, the main cause shown by this diagram is the continuous decrease of the free virtual disk amount. This could be related to an increase of the number of I/O requests to

the physical disk or to a decrease of the Swap Memory. In the last case, we conclude that the virtual allocated RAM is exhausted and more memory should be allocated to this virtual machine.

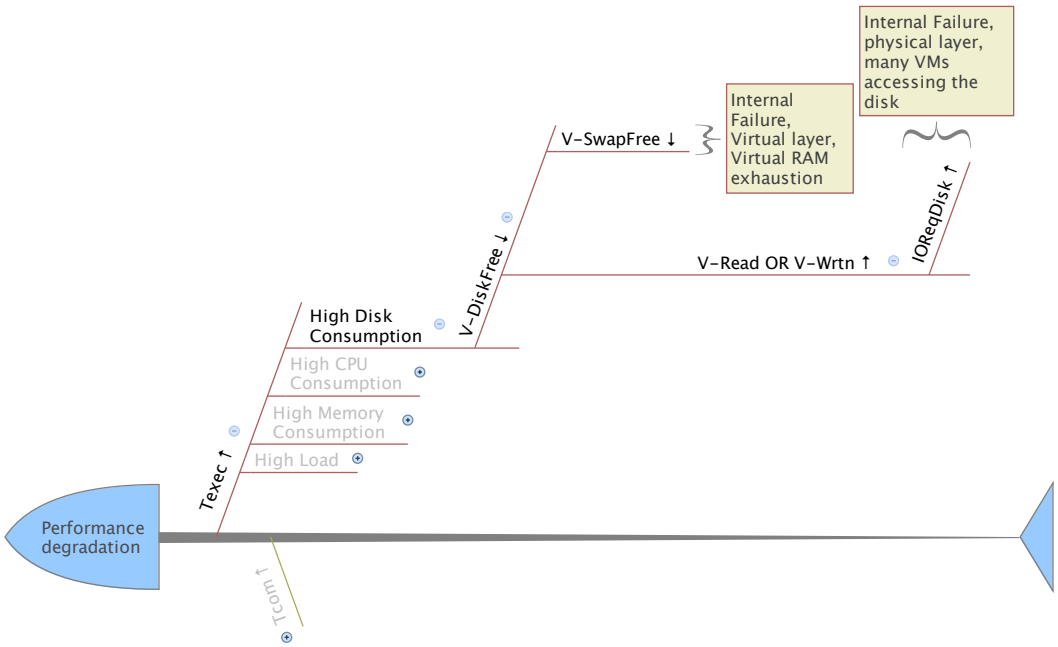

Figure 4.24: The analysis of an execution time degradation: The disk branch

Figure 4.19, 4.20, 4.21, 4.22, 4.23 and 4.24 constitute the different parts of the cause-effect diagram, which describes the different origins of a performance-related problem. This diagram is then transformed into a set of analysis rules that are implemented as queries within the CEP engine.

Each elementary branch of the cause-effect diagram is translated into an analysis rule. It is composed of a set of symptoms and a diagnosis report. An analysis rule is analytically represented by $\mathbf{R}(S_1, S_2, .. , S_n \,//\, D)$, where R is the rule; S_1, S_2, .. , S_n are the observed symptoms, and D is the deduced diagnosis report. For instance, the first part of the cause-effect diagram, showing the communication time analysis (see Figure 4.20) is translated into two analysis rules R_1 (see formula (4.4)) and R_2 (see Formula (4.5)).

$$\begin{aligned} R_1(Tcom\uparrow, P-BytesIn\downarrow Or P-BytesOut\downarrow \\ //External\ Failure, \\ Physical\ layer, \\ over-used\ links) \end{aligned} \tag{4.4}$$

$$R_2(Tcom \uparrow, V - BytesIn \downarrow OrV - BytesOut \downarrow, IOReqDisk \uparrow // Internal\ Failure, Physical\ layer, Many\ VMs\ accessing\ the\ disk) \quad (4.5)$$

The obtained rules are implemented as queries in the CEP engine. Therefore, we will get a large number of queries that could take a long time to be processed by the CEP engine. This could delay the recovery, especially in the context of a large Cloud computing environment. Thus, it is necessary to reduce the number of rules. Our rule reduction approach is detailed below.

Reduction of Rules

The cause-effect diagram (represented by Figures 4.19, 4.20, 4.21, 4.22, 4.23, 4.24) shows that some branches have exactly the same:

- first symptom
- last symptom
- diagnosis report

These branches can be reduced to a single branch, since they represent the specific cases of a general branch. The resulting branch is a very simple one, consisting of two symptoms (the first and the last one) and the diagnosis report.

The first part of the cause-effect diagram, dealing with the analysis of the communication time (see Figure 4.20) could not be reduced to a single branch, since the two last symptoms are different. Nevertheless, it is possible to replace six branches of the second part of the cause-effect diagram by only a single branch. The second part of the cause effect diagram deals with execution time analysis and is presented in Figures 4.21, 4.22, 4.23 and 4.24.

It is evident that six branches of this cause-effect diagram part have the same first symptom (the continuous increase of the execution time: Texec ↑), the same last symptom (the continuous increase of the number of the I/O requests to the physical disk: IOReqDisk ↑), and the same diagnosis report. These six branches are shown in Figure 4.25.

The diagram of Figure 4.26 shows the resulting branch, replacing the six branches represented in Figure 4.25.

The new branch shown in Figure 4.26 is translated into a simple analysis rule. The resulting rule replaces six analysis rules and is described by Formula (4.6).

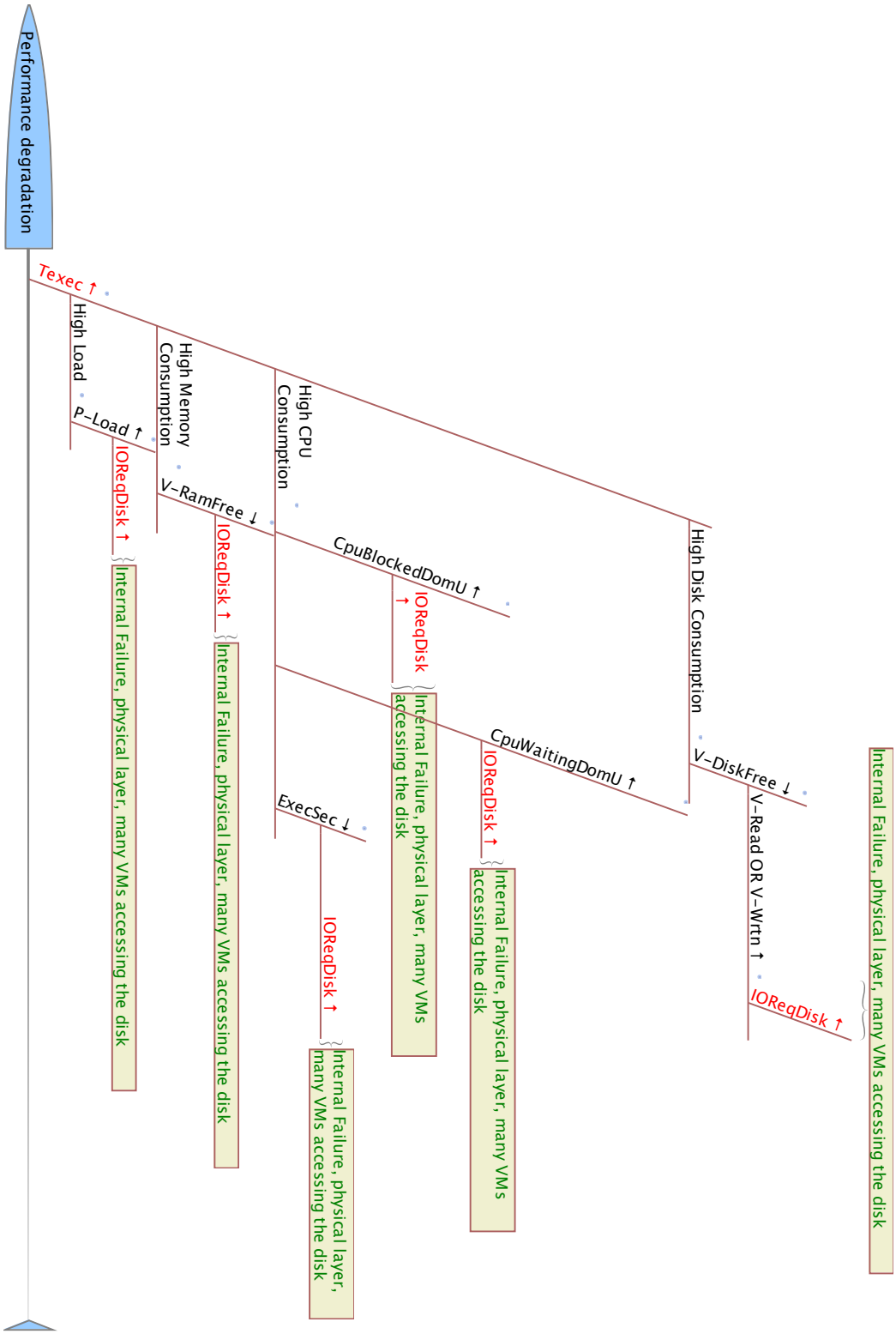

Figure 4.25: Similarities between six branches in the cause-effect diagram

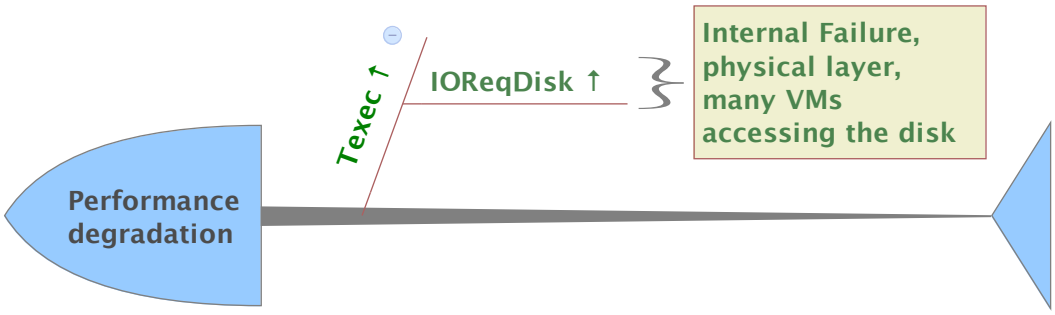

Figure 4.26: A simple branch (analysis rule) replaces 6 branches (6 analysis rules).

$$R_s(Texec \uparrow,\ IOReqDisk \uparrow //\\ Internal\ Failure,\\ Physical\ layer,\\ Many\ concurrent\ VMs\ accessing\ the\ disk) \tag{4.6}$$

This section has presented the first version of the monitoring and analysis agent, CEP4CMA. It fulfills the third requirement **R3**, but still suffers from some drawbacks. Actually, it is based on the use of a single CEP engine that can easily become a bottleneck. Therefore, we defined a novel dynamic architecture for Cloud performance monitoring and analysis based on the use of many CEP engines. The proposed architecture is called D-CEP4CMA and is detailed in Section 4.4.2.

4.4.2 D-CEP4CMA

This section presents our dynamic CEP approach for Cloud performance monitoring and analysis, called D-CEP4CMA for "**D**ynamic **C**omplex **E**vent **P**rocessing for **C**loud **M**onitoring and **A**nalysis". The basic idea is to dynamically switch between different CEP architectures depending on the current conditions of the observed Cloud environment. D-CEP4CMA is deduced from an experimental study of three different CEP architectures for Cloud monitoring and analysis, a centralized one and two distributed ones. First, we outline its general architecture. Second, the CEP architectures involved in the D-CEP4CMA life cycle are presented. Finally, we describe the D-CEP4CMA algorithms that allow us to switch between the different CEP architectures.

D-CEP4CMA Architecture

Figure 4.27 shows the architecture of D-CEP4CMA. The analysis agent of D-CEP4CMA is based on the use of many CEP engines and dynamically switches

between different CEP architectures. The used CEP architectures are described below.

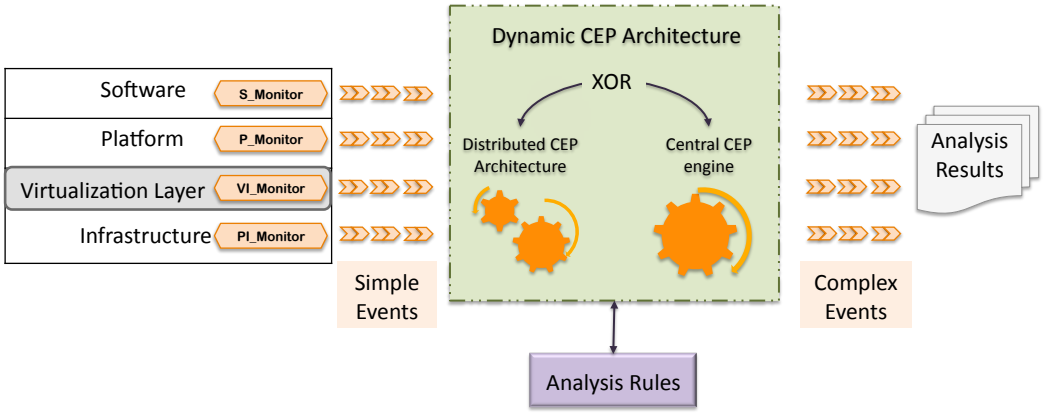

Figure 4.27: The architecture of D-CEP4CMA

The CEP Architectures

In this section, three CEP architectures for Cloud monitoring and analysis, a centralized architecture relying on a single CEP engine and two distributed architectures based on a set of cooperating CEP engines, are presented.

A Centralized CEP Architecture Figure 4.28 shows our centralized architecture for Cloud monitoring and analysis. It is based on a single CEP engine. The CEP engine processes all monitored data and detects performance problems in the Cloud using our analysis rules. The analysis rules are implemented as EPL queries within the Esper CEP engine.

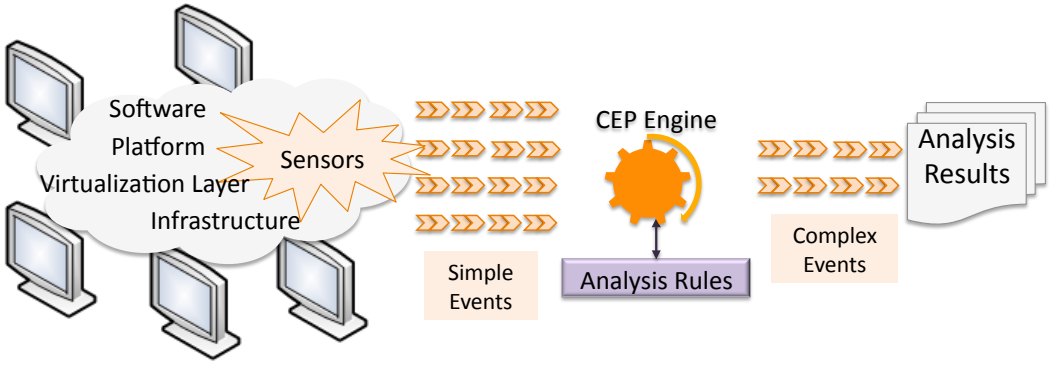

Figure 4.28: The centralized CEP architecture

The centralized CEP architecture suffers from two principal problems: a) it has to process a potentially large number of events and thus may become

bottleneck; b) it represents a single point of failure. Therefore, it is not suitable for large Cloud computing environments. Consequently, D-CEP4CMA involves two other distributed CEP architectures to deal with the case of large Clouds. They are presented below.

Distributed CEP Architectures Figures 4.30 and 4.31 show two distributed CEP architectures for Cloud monitoring and analysis. They rely on "many" CEP engines instead of a single one. The used CEP engines have two different roles and are able to communicate with each other. The first role is called "**CEP Manager**". It is taken by the CEP engine that processes the main analysis rules[4]. Depending on the size of the Cloud, the distributed CEP architectures make use of one or several CEP Managers(s). If several CEP Managers are present, each CEP Manager is responsible for a particular set of physical Cloud machines, and each of them operates independently without communicating with other CEP Managers. The second role is called "**CEP Worker**". It is played by the rest of CEP engines. The CEP Workers have two main tasks. First, they share the analysis tasks. Second, they filter events and only send pertinent events to the CEP Manager. The first task is related to the distributed nature of the analysis, i.e., the monitoring data is not sent to a single CEP, but to many cooperative CEP Workers. The second task concerns the functionality of filtering events. It is based on our Outlier Detector approach. When a CEP Worker detects an outlier, it notifies the rest of CEP Workers and the monitoring sensors. The latter sends data to the CEP Manager(s). The Outlier Detector approach is presented below.

The Outlier Detector As shown in Figure 4.29, the Outlier Detector is based on robust statistics to calculate the z-score and detect outliers. Its life cycle consists of two phases. The first phase is the training period. During this phase, the Outlier Detector computes the values of the median and the Mean Absolute Deviation (MAD) of its inputs. To calculate the median of a set of n values (x_i), we sort them and calculate $x_{\frac{n+1}{2}}$ if n is odd, and $\frac{x_{n/2}+x_{n/2+1}}{2}$ if n is even [86]. The MAD is the median of all absolute deviations, multiplied by a corrective coefficient (see Formula (4.7)) [86]. The median and the MAD are more robust against outliers than the mean and the standard deviation (SD), respectively [86].

$$MAD = 1.483 * median_{i=1..n}|x_i - median(x_j)_{j=1..n}| \tag{4.7}$$

The second phase is used to compute the z-score of the incoming monitored data (x_i). The z-score of x_i is the difference between x_i and the median, divided by the MAD (see Formula (4.8)) [86].

[4]In the centralized CEP architecture, the CEP Manager corresponds to the used (single) CEP engine.

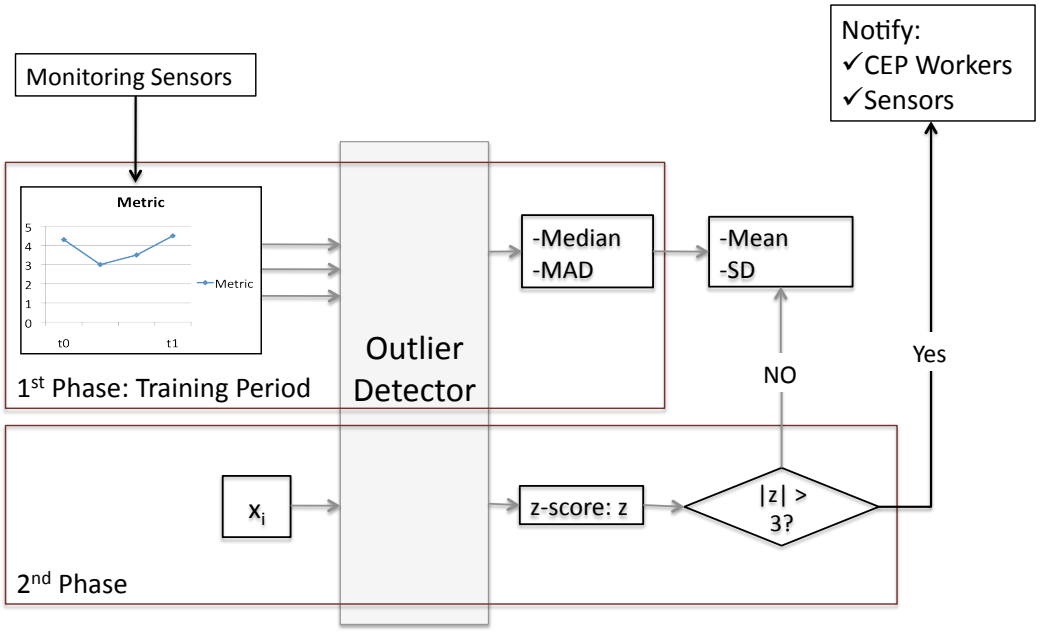

Figure 4.29: The outlier detector

$$\text{z-score} = \frac{x_i - median(x_j)_{j=1..n}}{MAD} \tag{4.8}$$

The z-score indicates whether the new incoming monitored value x_i is an outlier. In fact, if the absolute value of the z-score exceeds 3, then x_i is an outlier. Otherwise, x_i is an acceptable value [26]. If ($|\text{z-score}| < 3$), the Outlier Detector computes the mean and the standard deviation, while using the new incoming value and the old computed median and MAD. The mean and the standard deviation are used in the second phase, since we are sure that the used values are not outliers. The use of robust statistical metrics (like the median and MAD) during the first phase is required, since outliers could often appear during the training period. However, it is more significant to use traditional statistical metrics during the second phase, since we are sure that all values, used to calculate the mean and the standard deviation, are not outliers. In fact, during the second phase we calculate the z-score of every new incoming data x_i, and check whether x_i is an outlier. If x_i is not an outlier, we use it to update the mean and the standard deviation. Otherwise, we keep the old values of the mean and the standard deviation (see Figure 4.29). Moreover, the mean and the standard deviation are more precise than the median and the MAD, respectively, in the absence of outliers.

The two distributed CEP architectures make use of the Outlier Detector. Their design and functionality are described below.

Design I Figure 4.30 shows the first distributed CEP architecture based on multiple CEP engines. It is called "Design I" in the remainder of this thesis. Design I assigns a CEP Worker to every Cloud machine (physical and virtual): one CEP Worker per Cloud machine. The CEP Workers are running on all phys-

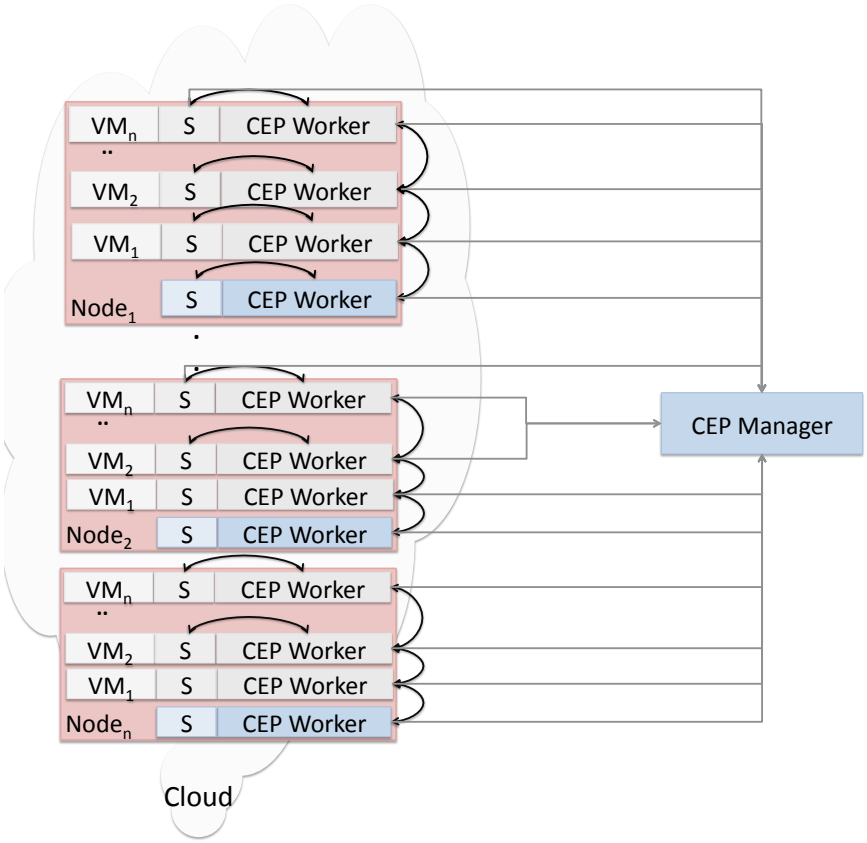

Figure 4.30: The distributed CEP architecture: Design I

ical and virtual machines of the Cloud. They communicate with the monitoring sensors (S) and their CEP Worker neighbors. First, the sensors send the values of the monitored metrics to the CEP Workers. Since the CEP Workers implement the Outlier Detector, they process the received data and check whether there are outliers. If a CEP Worker detects an outlier, it first sends the recorded data (at instant t_i) to the CEP Manager, which is in charge of processing the analysis rules. Second, it notifies its CEP Worker neighbors to ask them to send the last monitored data (recorded at t_i) to the CEP Manager. Third, the CEP Worker notifies the sensors and asks them to send the next N data values to the CEP Manager who processes the main analysis rules (see Figure 4.30). N depends on the used analysis rules. It indicates how often the symptoms should be observed to detect an unwanted situation. Therefore, N represents the number of required data values to process the analysis queries. Design I makes use of a few Out-

lier Detectors, related to pertinent metrics, and implements a selection algorithm that allows to only send pertinent data, regarding the detected outlier.

Design II Figure 4.31 depicts the second distributed architecture, called Design II. It involves less CEP Workers: one CEP Worker per physical node. The CEP Worker processes data coming from the physical node and its virtual machines. If one of the CEP Workers detects an outlier, it sends and asks sensors / CEP Workers neighbors to send monitored data to the CEP Manager, as in Design I. Design II assigns Outlier Detectors to pertinent metrics, and implements a selection algorithm to send selected data to the CEP Manager.

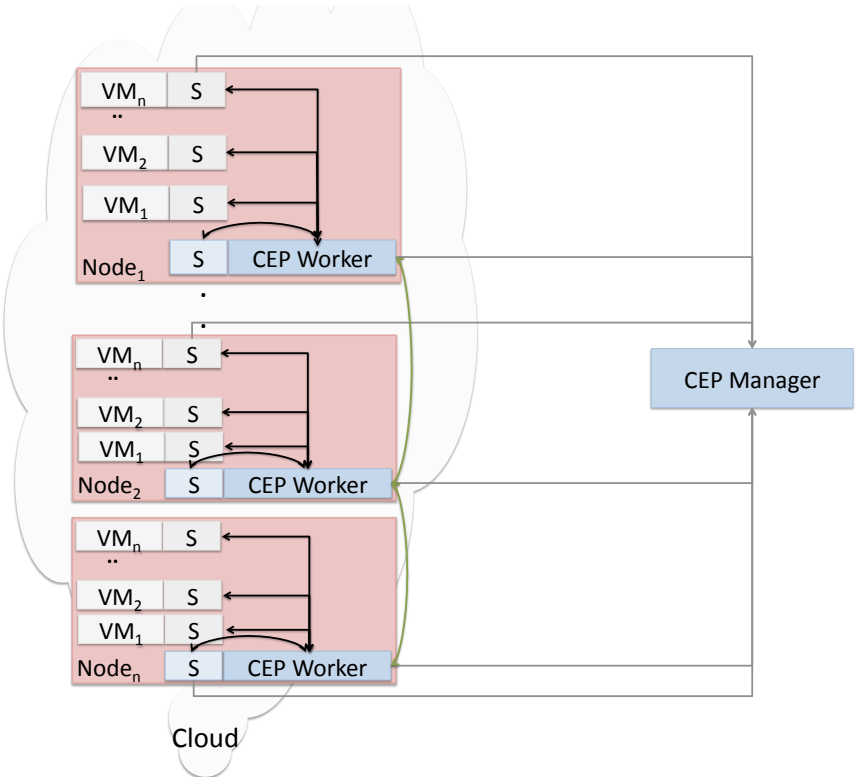

Figure 4.31: The distributed CEP architecture: Design II

The centralized architecture, Design I and Design II have been experimentally evaluated. The evaluation was based on:

- Quality analysis metrics, such as precision and recall.
- Performance indicators, such as load and network communication, of the Cloud machines.
- Performance indicators of the CEP Manager, such as the load and the memory of its hosting machine.

The choice of these evaluation criteria is partially based on the relationships between them and the Cloud size. Indeed, the CEP Manager machine memory and load metrics depend on the volume of events (i.e., monitoring results) processed by the CEP Manager. The volume of these events is highly related to the Cloud size. In fact, the number of monitored events increases when the number of Cloud machines increases.

The lessons learned from the experimental evaluation can be summarized as follows:

- The centralized architecture is a single point of failure, but it gives good results in terms of precision and recall. The centralized architecture is also more efficient than the distributed architectures, since it neither saturates the network nor the virtual machines. Thus, a centralized architecture should be used when a single CEP engine is able to handle all received data.

- A distributed architecture is more efficient than the centralized CEP architecture for large scale Cloud environments.

The evaluation results illustrate that from a performance point of view, we should not use a distributed CEP architecture, if a centralized CEP architecture could solve the problem. They also indicate that it is necessary to migrate to a distributed architecture, if the amount of data to be processed by the CEP engine exceeds its capacities, in terms of load and used memory.

These conclusions have been used to propose the dynamic CEP architecture algorithms presented below.

Algorithms for a Dynamic CEP Architecture

The basic idea of the dynamic architecture is to take profit of the centralized and distributed architectures and avoid their disadvantages, while choosing the most suitable design. It is mainly based on the scale up and scale down algorithms shown in Figures 4.32 and 4.33, to decide whether to activate particular architectural components.

The scale up algorithm (see Figure 4.32) allows us to choose the suitable architectural design when the size of the Cloud grows. It starts with a centralized CEP architecture relying on a single CEP engine. The scale up algorithm periodically checks whether the single CEP engine is overloaded, by comparing the load and the free memory of the physical machine hosting the CEP engine to the upper thresholds. If it is overloaded (in terms of memory or load), the algorithm switches to Design II, and activates the required number of components for Design II. It then checks whether Design II leads to an overloaded physical machine hosting the CEP Manager. If this is the case, the algorithm switches to Design I and activates the selection algorithm. Otherwise, the algorithm checks

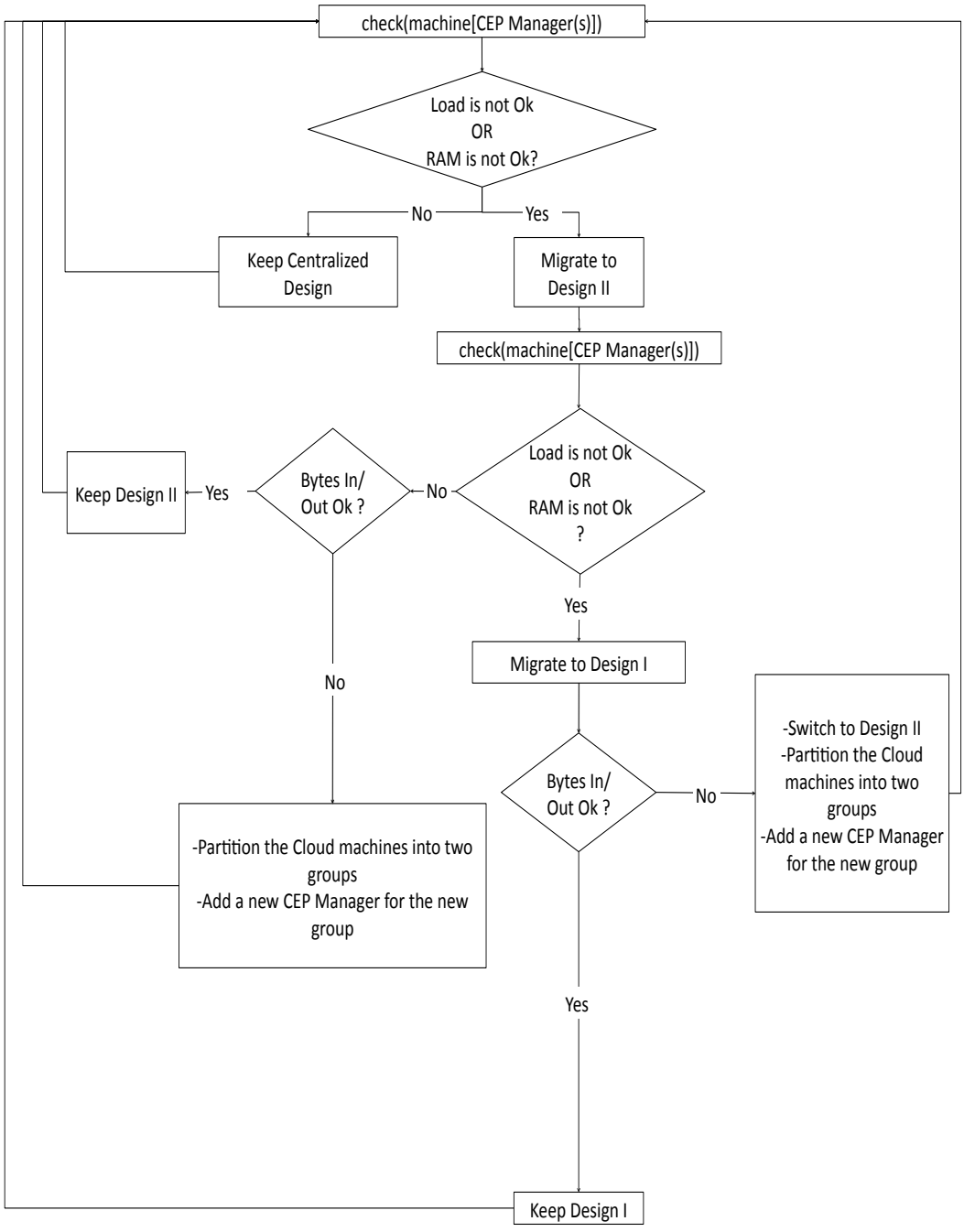

Figure 4.32: D-CEP4CMA algorithm: scale up

the network state by comparing bytes In/Out to an adequate threshold. Actually, the average of the bytes In/Out of the Cloud machines are measured, to assess the network state of the Cloud. If the network state is acceptable, then the algorithm keeps Design II. Otherwise, it divides the physical Cloud machines

into two groups and starts a new CEP Manager for the new group, following the centralized design. In the case the algorithm has migrated to Design I, it checks the network state by measuring the bytes In/Out. If the network state is acceptable, it keeps Design I. In the opposite case, it switches to Design II, partitions the physical Cloud machines into two groups and activates a new CEP Manager for the new group. The partitioning process is based on two key parameters: CEP_Id and Part_Id. CEP_Id identifies the CEP Manager. It is equal to the IP address of its hosting machine. Every Cloud machine is identified by (CEP_Id, Part_Id), while Part_Id designates the partition number. It is either equal to 1 or to 2. If the switching algorithm decides to partition the Cloud machines into two groups, the machines with Part_Id equal to 2 will be assigned to the new CEP Manager. Their CEP_Id value is equal to the IP address of the machine hosting the new CEP Manager. The set of machines assigned to the old / new CEP Manager are divided into two groups. The Part_Id of the first group is equal to 1, while the Part_Id of the second group is equal to 2.

As shown in Figure 4.33, the scale down algorithm allows us to choose the suitable architectural design when the size of the Cloud decreases. It periodically checks whether the number of used CEP Managers is greater than 1. If only one CEP Manager is used, the algorithm checks whether the centralized design is currently adopted. If the centralized design is not adopted, the scale down algorithm checks whether the CEP Manager machine is under-used, by comparing the load and free memory values to the lower thresholds. If the CEP Manager machine is under-used, the algorithm migrates to the centralized design. Otherwise, it keeps the current design. If there are many CEP Managers (more than one), the scale down algorithm checks the load and the free memory values of all used CEP Manager machines. If it detects that there are under-used CEP Manager machines, the scale down algorithm checks their currently adopted design. If there is more than one CEP Manager following the centralized design, the scale down algorithm replaces all pairs of CEP Managers by a single CEP Manager. If the centralized design was not adopted, the scale down algorithm migrates to the centralized design.

Ganglia is used to measure the load and the free memory metrics of the machine hosting the CEP Manager. The load and the free memory thresholds are defined as follows. In both cases, the upper thresholds are equal to the sum of the corresponding mean value and the standard deviation. The lower threshold of the load is equal to the minimum (observed) value of the load minus the standard deviation. The lower threshold of the memory is equal to the maximum (observed value) of the memory plus the standard deviation. The mean, the minimum, the maximum and the standard deviation are obtained by measuring the load and free memory values of the machine running the CEP Manager in normal conditions.

The checking period depends on:

- The size of the Cloud

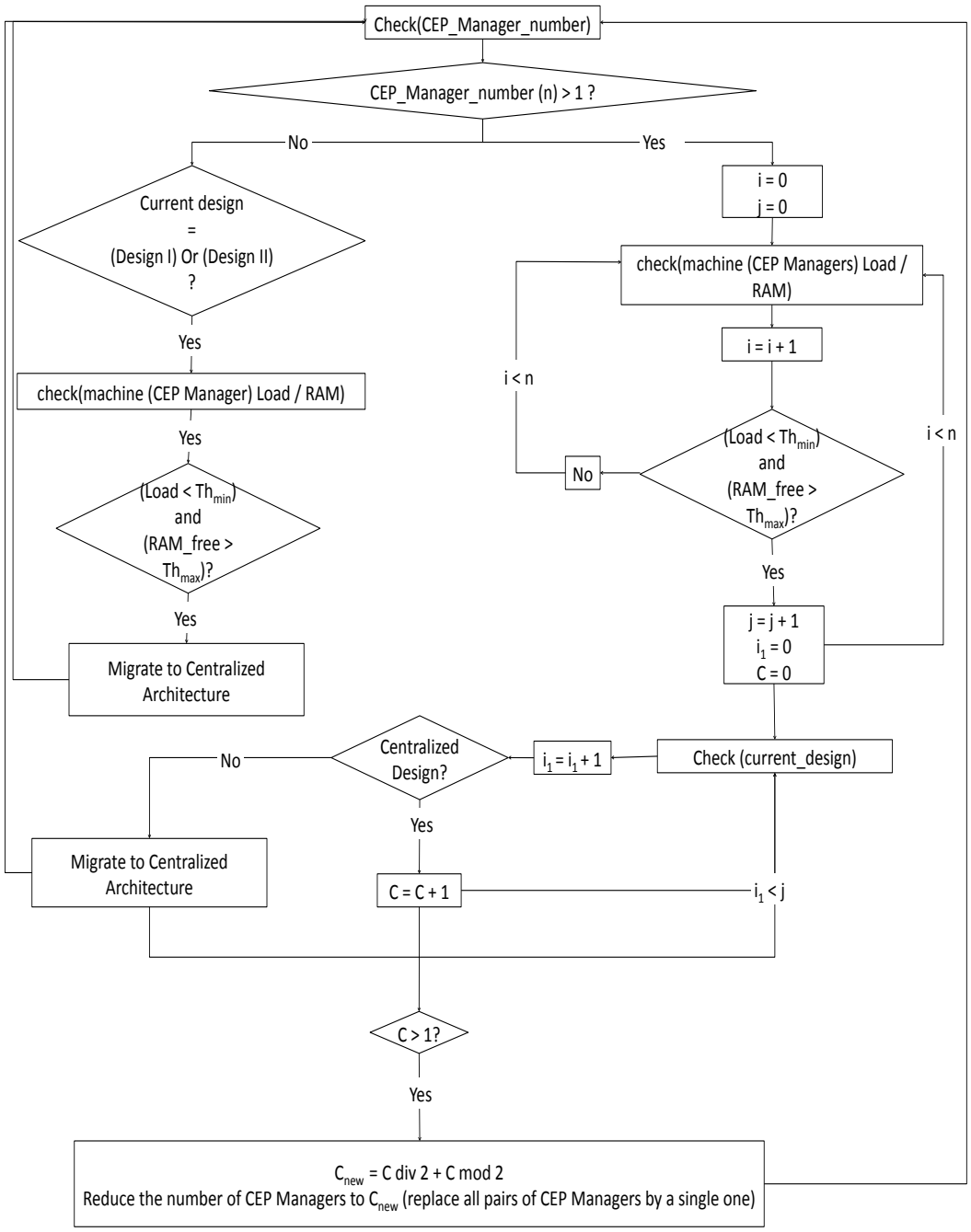

Figure 4.33: D-CEP4CMA algorithm: scale down

- The frequency of receiving monitoring data
- The user preferences

A short checking period implies a faster decision. However, a long period leads to a more accurate decision.

It should be pointed out that the D-CEP4CMA algorithms (scale up and scale down) stop their operations for a period of time (WT) when a new CEP architecture is selected. This allows us to test the efficiency of the newly selected CEP architecture. If the performance indicators of the CEP Manager have improved after the end of the WT period, then the selected CEP architecture is the best one for this Cloud and should be kept until the size of the Cloud changes. Otherwise, the D-CEP4CMA algorithms resume their search for a suitable CEP architecture. The WT period is the time needed by the machine running the CEP Manager to reach its normal behavior after a degradation of its performance parameters. In this work, the WT period is experimentally measured.

D-CEP4CMA fulfills the requirement **R4**, in ensuring the scalability (scale-up and scale-down) of the analysis approach.

The outputs of D-CEP4CMA (i.e., diagnosis reports) are used by the action manager framework, to identify and apply the adequate recovery action. The next section describes the action manager framework.

4.5 The Action Manager Framework

This section presents the novel action manager framework. It is used to fix performance-related problems that might occur in Cloud computing environments. First, we outline its general architecture. Then, the main components of the action manager framework are described.

Action Manager Architecture: Figure 4.34 depicts the architecture of the action manager framework.

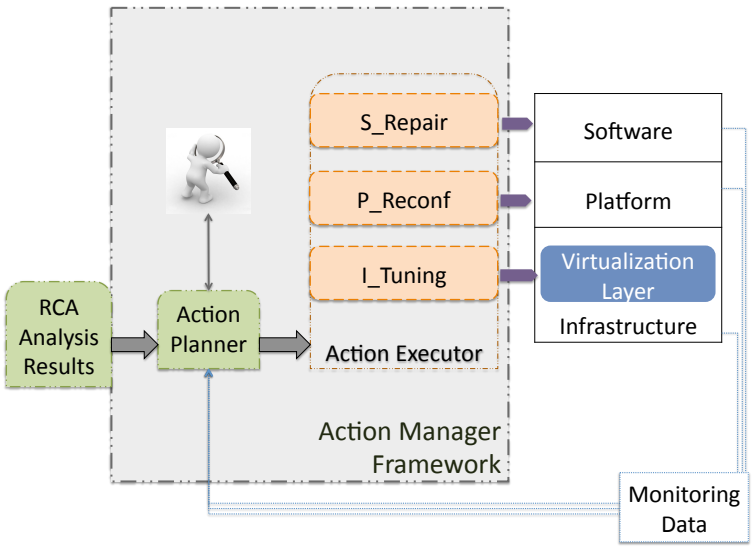

Figure 4.34: The action manager framework

It consists of two main components: the action planner and the action executor. It follows three steps to rectify a performance-related problem. First, the action planner uses the received diagnosis reports to choose the adequate recovery action. Second, the chosen recovery action will be applied by the action executor. Third, the action planner checks whether the applied action has successfully repaired a performence-related problem in the Cloud. In case the applied action has failed to repair the problem, the action planner looks for another repair action. If another recovery action does exist, the action planner applies it. Otherwise, it asks the Cloud administrator to update the set of recovery actions related to this performance degradation.

As shown in Figure 4.35, each performance-related problem is identified by:

- A set of suitable recovery actions
- The last observed symptom in the path of the corresponding fishbone branch

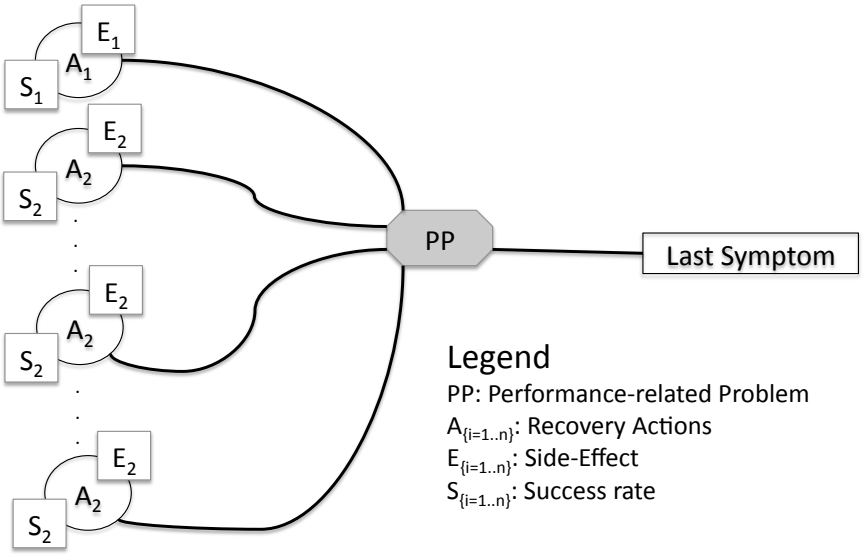

Figure 4.35: A performance-related problem: The model

Each recovery action is characterized by (1) its side-effect level ($E_{i,i=1..n}$) and (2) its success rate ($S_{i,i=1..n}$). The side-effect level of a recovery action describes the severity of this action. Its value varies between 0 and 2. If it is equal to 0, this means that this recovery action has no effect. In case the side-effect level is equal to 1, this indicates that this recovery action has an acceptable effect on this system. The recovery action has a severe effect on the system, if the side-effect value is equal to 2. For instance, "Kill a Virtual Machine" is a severe recovery action. Its side-effect level is equal to 2.

The action planner and the action executor are described below.

The Action Planner

The action planner is in charge of selecting the suitable recovery action and validating its success. It is based on our novel repair algorithm (see Figure 4.36).

As shown in Figure 4.36, the repair algorithm starts by sorting the available recovery actions according to their side-effect level (E). Afterwards, the algorithm checks whether the recovery action having the minimum value of E is not severe ($E < 2$). If this recovery action is not severe, the repair algorithm applies it and validates its success, by checking whether the last symptom has disappeared. If the last symptom has disappeared, the repair algorithm increases the success rate of this recovery action by 1. Otherwise, it checks the side-effect of the next recovery action if there is one; and decreases the success rate of the failed recovery action by 1. In case the side-effect of the first selected recovery action is equal to 2, the repair algorithm checks whether there are more recovery actions associated with the corresponding performance-related problem. If there are more recovery actions, the algorithm selects the best one in terms of success rate ($S > 0$), applies it and validates it, using the same procedure as described above. If there is only one recovery action with a side-effect equal to 2 and a success rate equal to 0, the repair algorithm asks the Cloud administrator to decide whether to apply this recovery action. If the Cloud administrator approves this recovery action, it is retained and its success rate is updated. Otherwise, this recovery action is removed and will be substituted by an "emergency" recovery action identified by the Cloud administrator.

The Action Executor

The action executor is in charge of applying the chosen recovery action. It is composed of three execution modules: I_Tuning, P_Reconf and S_Repair. They are described below.

I_Tuning executes recovery actions on the virtualization layer, such as tuning resources, killing virtual machines or migrating them from a physical node to another.

P_Reconf operates on the platform layer. It executes platform reconfiguration actions such as re-setting the number of concurrent clients and killing processes.

S_Repair acts on the software layer and applies related recovery actions, such as substituting and duplicating services.

The action manager framework fulfills the requirement **R5**. In fact, it assigns many recovery actions to a given performance-related problem and checks the success of the applied action.

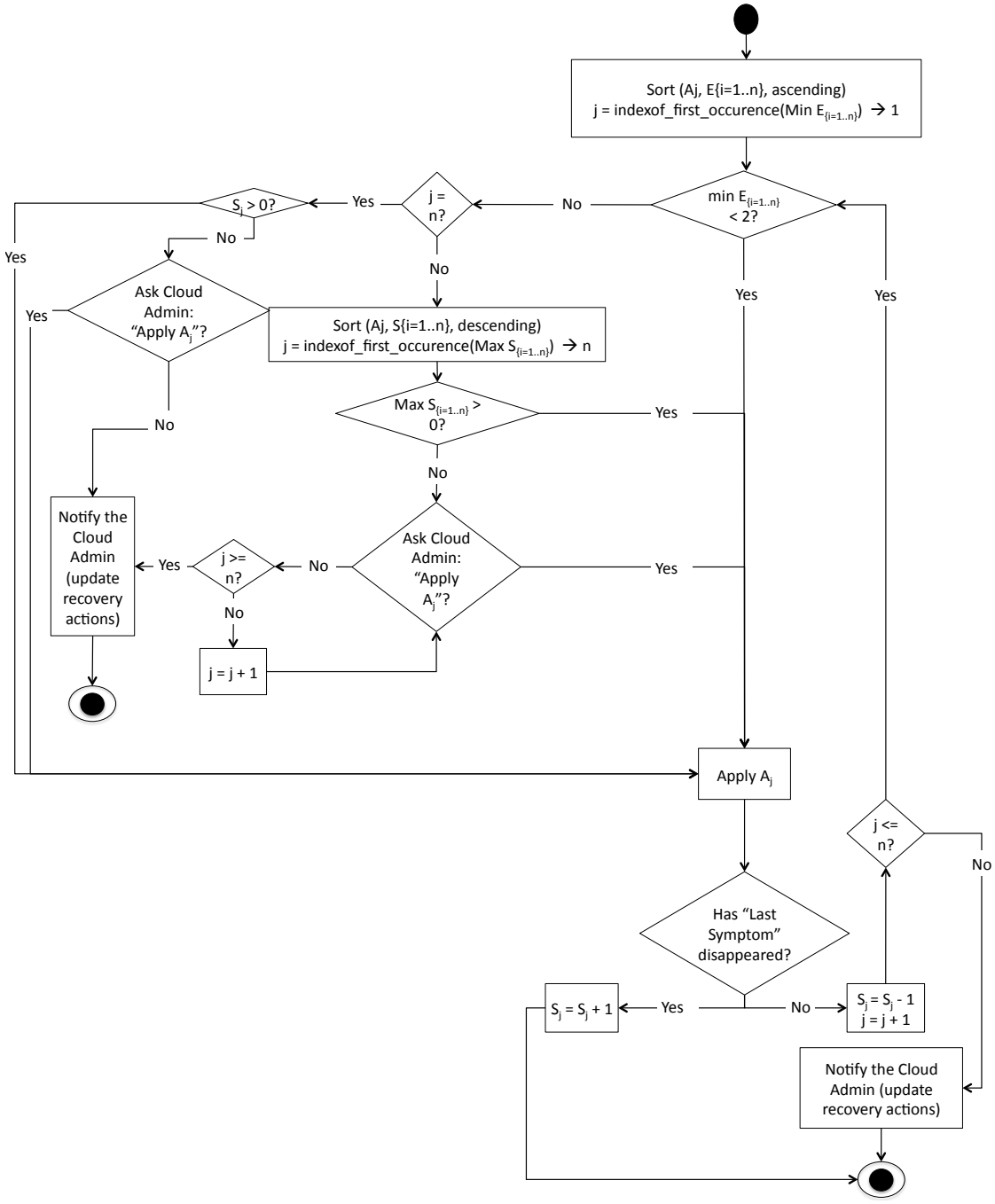

Figure 4.36: The repair algorithm

4.6 Summary

This chapter has presented a cross-layer reactive performance monitoring approach for Cloud computing environments, called CEP4Cloud. It allows us to monitor and analyze performance metrics across Cloud layers, detect performance-related problems and fix them with minimal human intervention. This chapter has demonstrated that CEP4Cloud fulfills all requirements stated in the third chapter (see Section 3.5). Implementation details regarding CEP4Cloud and its main components are given in Chapter 5.

"Talk is cheap. Show me the code."

Linus Torvalds

5 Implementation

5.1 Introduction

This chapter details the implementation of CEP4Cloud and its main components. First, it presents a high level view of CEP4Cloud, while showing its main structure. Then, it gives implementation details regarding the components of CEP4Cloud: the multi-layer monitoring agent, the cross-layer analysis agent and the action manager framework. Parts of this chapter have already been published in [64–67, 69].

5.2 Implementation of CEP4Cloud

CEP4Cloud makes use of the Esper-4.6.0 CEP engine (Java version). Therefore, the structure of CEP4Cloud consists of three main packages: sensors, analysis and sinks (see Figure 5.1). The package "sensors" is in charge of establishing the communication between the multi-layer monitoring agent and the CEP engine. The package "analysis" implements the main functionality of the CEP engine (i.e., analysis rules). The package "sinks" is used by the action manager framework to launch suitable recovery actions. The packages "dynamicalgo" and "action.manager" are used by the analysis agent and the action manager framework, respectively.

CEP4Cloud is based on three main steps. In the first step, we register the sources of events to allow the communication between our multi-layer monitoring agent and the CEP engine. Listing 5.1 illustrates the first step. It shows three examples of event sources registration.

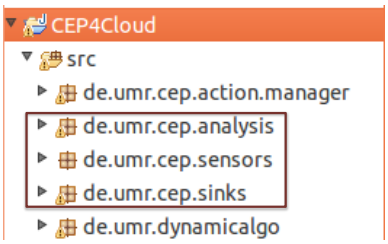

Figure 5.1: The structure of CEP4Cloud

```
cepConfig.addEventType("IoStatData", IoStatSensor.class.
   getName());
cepConfig.addEventType("JVMData", JVMSensor.class.getName()
   );
cepConfig.addEventType("LoadData", LoadSensor.class.getName
   ());
```

Listing 5.1: Registration of event sources

The second step consists of implementing and launching our analysis rules as EPL queries. This allows us to process the received event sources and characterize the state of the Cloud. Listing 5.2 presents the implementation / launching procedure of the analysis rules. A concrete example of the implementation of an analysis rule is given in Listing 5.13.

```
EPStatement IOStatement = cepAdm.createEPL(analysis_rule);
```

Listing 5.2: The implementation of an analysis rule

The last step deals with the implementation of the event sinks and their registration within the EPL queries. In our case, the event sinks allow us to launch the suitable repair action, if a performance-related problem has been detected by the EPL queries. Listing 5.3 presents a code snippet allowing us to register the EPL query of Listing 5.2 within the sink "RepairIO". The latter implements our action manager framework to repair an I/O performance-related problem.

```
IOStatement.addListener(new RepairIO());
```

Listing 5.3: Registration of event sinks

The first task of CEP4Cloud consists of collecting Cloud metrics and sending them to the CEP engine. Section 5.3 gives details regarding our multi-layer monitoring agent.

5.3 Monitoring

The multi-layer monitoring agent is composed of four main components: S_Monitor, P_Monitor, VI_Monitor, PI_Monitor. Implementation details regarding these components are given, below.

5.3.1 S_Monitor: AOP4CSM

AOP4CSM has been implemented within Axis1 and Axis2. The Axis1 implementation is presented in the following; the Axis2 implementation works accordingly.

The implementation within Axis1 makes use of two components. The first one, called **AOP4CSM Client**, operates at the client side and calculates (1) the response time and (2) the number of invocations (all invocations: advice 1 and successful invocations: advice 4). The second component operates at the server (provider) side. It evaluates the execution time. It is called **AOP4CSM Server**. The implementation of **AOP4CSM Client** and **AOP4CSM Server** is described below.

Implementation of AOP4CSM Client

AOP4CSM Client is aspect code that intercepts the Client at t_1 and t_4. Its implementation is based on the identification of the methods that the web service engine invokes at t_1 and t_4. In Axis1 [5], the method invoked at t_1 is *init(...)* of the *AxisEngine* class located in *org.apache.axis* package (see Listing 5.4 line 6). When the client sends a request, the Axis engine of the client side is invoked via the method *init(...)*. At t_4, the method *extractAttachments(...)* of the *Stub* class located in *org.apache.axis.client* package (see Listing 5.4 line 11) is invoked. The method *extractAttachments(...)* implies, when invoked, that the response is received by the client.

```
1 //Calculate the number of successful invocations
  int InvosNumber = 0
3 //Pointcut1: t1
  pointcut requestClient(): call(* execution(* org.apache.
     axis.AxisEngine.init(..));
5 after(): requestClient() {
  Treq = System.currentTimeMillis();
7 }
  //Pointcut4: t4
9 pointcut responseClient(): execution(* org.apache.axis.
     client.Stub.extractAttachments(..));
  after(): responseClient() {
11 Tresp = System.currentTimeMillis();
  RespTime = Tresp - Treq;
13 //send and/or save RespTime
  // Increment the number of invocations
15 InvosNumber++;
  }
```

Listing 5.4: AOP4CSM client implementation for Axis 1

Implementation of AOP4CSM Server

The second component of our monitoring approach is the AOP4CSM Server. It is aspect code (see Listing 5.5) that assesses the execution time. To implement the AOP4CSM Server within Axis1, it is necessary to identify pointcut 2 and pointcut 3 and the methods intercepted. Pointcut 2 corresponds to the instant t_2 when the request arrives at the server side. Pointcut 3 describes the instant t_3 when the response leaves the server side. The execution of the method *invoke(...)* of the class *SOAPService* located in the *org.apache.axis.handlers.soap* package is in charge of the request processing at the server side. This means that saving the instants before and after the execution of this method lead to the computation of the execution time value (see Listing 5.5).

Thus, pointcuts 2 and 3 correspond to the interception of the execution of this method, and the related advices should be applied before and after this method, respectively (see Listing 5.5).

```
// ...
//Pointcut2: t2
pointcut requestServer(): execution(* org.apache.axis.
   handlers.soap.SOAPService.invoke(..));
before(): requestServer() {
TreqS = System.currentTimeMillis();
}
//Pointcut3: t3
pointcut responseServer(): execution(* org.apache.axis.
   handlers.soap.SOAPService.invoke(..));
after(): responseServer() {
TrespS = System.currentTimeMillis();
Texec = TrespS - TreqS;
//Send and/or save Texec
}
// ...
```

Listing 5.5: AOP4CSM server implementation for Axis 1

AOP4CSM also handles multiple clients. It is able to distinguish between clients by using their IP addresses. Moreover, AOP4CSM uses the request references to differentiate between concurrent requests running on the same client. Thus, it associates monitored QoS parameters to the corresponding client. Based on AOP, the AOP4CSM Server extracts the client and the server IP addresses that correspond to the collected monitored QoS parameters. Furthermore, the AOP4CSM Client distinguishes between concurrent requests running on the same client, while extracting their references.

It should be pointed out that the AOP4CSM components (Server and Client) are completely independent of the original server (Axis in our case). In fact, the developed aspect codes are not located inside the server source code. Thanks to

the weaving mechanism of AOP, they intercept methods at defined join points and record relevant timestamp information without modifying the source code of Axis.

AOP4CSM Installation Procedure

The installation of AOP4CSM does not need any access to the source code of the service and can be completely performed by the Cloud client as long as the client has access to the hosting platform, i.e., the web service middleware. Thus, the client only has to upload the AOP4CSM components to the client side (AOP4CSM Client component) and to the server side (AOP4CSM Server component). As shown in Figure 5.2, the installation of AOP4CSM is quite simple.

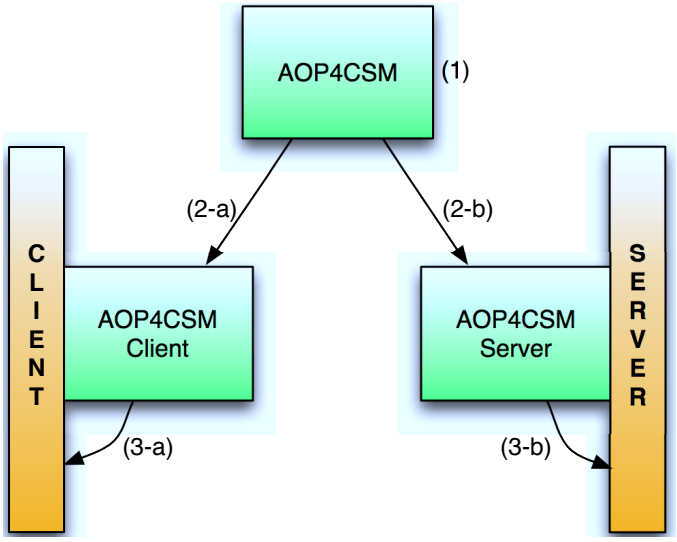

Figure 5.2: AOP4CSM: Installation process

It is composed of three main steps. The first step (1) consists of downloading the implementation of AOP4CSM. The second step distributes the components of AOP4CSM to the server and the client sides. Thus, it consists of two sub-steps. In the first sub-step (2-a), the Cloud client uploads the AOP4CSM Client to the client side. The second sub-step (2-b) consists of uploading the AOP4CSM Server to the server side. Finally, in the third sub-steps (3-a) and (3-b), the Client enables the weaving in the client and the server side while launching traditional weaving commands, i.e., it takes care that the corresponding advice codes will be executed at the defined join points. The weaving mechanism does not need any access to the source code; it is applied to the executables of the system (.jar files of Axis in our case). In fact, the weaving commands only need access to the source code of AOP4CSM, which is completely independent of the server source code.

AOP4CSM has also been implemented within Axis2. Details about the implementation of AOP4CSM within Axis2 are given on the AOP4CSM web site [2]. The most important task for implementing AOP4CSM in other environments consists of identifying the methods to be intercepted at the four instants and their corresponding join points. This task is quite simple, since it only requires knowledge about the server structure.

It should be pointed out that as SaaS provider, we operate at the Platform layer. So, we have access to the middleware. Thus, it is possible to install AOP4CSM on commercial PaaS solutions. Moreover, AOP4CSM can be installed in the context of commercial SaaS Solutions if the cloud provider allows an access to the executables of the hosting platform.

5.3.2 P_Monitor: JVMSensor

JVMSensor allows us to collect JVM-related parameters. It is written in Java and based on the Jconsole tool. This section gives some details regarding the implementation of JVMSensor. It focuses on:

- The source code allowing us to collect thread-related metrics.
- The source code allowing us to collect memory metrics such as the heap memory usage.

Collect thread-related metrics: Source code

Listing 5.6 shows a code snippet allowing us to collect thread-related parameters.

```
private List<JvmThread> getThreadList() {
  List<JvmThread> threadList = new ArrayList<JvmThread>();
  try {
            ThreadMXBean tb = jmxConnector.getBean(
               ThreadMXBean.class, ManagementFactory.
               THREAD_MXBEAN_NAME);
            for (long threadId : tb.getAllThreadIds()) {
                threadList.add(fillJvmThread(tb.
                   getThreadInfo(threadId),
                tb.getThreadCpuTime(threadId)));
            }
        } catch (BeanNotFoundException e) {
            logger.warn("Cannot get thread bean from JVM. "
                + e);
            return null;
        }
  return threadList;
 }
```

Listing 5.6: Thread-related metrics

As shown in Listing 5.6, JVMSensor follows four steps to collect thread metrics. First, it connects to the JMX agent (see Line 4 of Listing 5.6). Second, it gets the identifier of all threads (see Line 5 of Listing 5.6). Third, it builds the list of all threads (see Line 6 of Listing 5.6). Finally, it returns the thread list (see Line 13 of Listing 5.6).

Collect (jvm) memory metrics: Source code

Listing 5.7 shows a code snippet allowing us to collect (jvm) memory parameters.

```
1 public void getMemoryStats() throws BeanNotFoundException {
      MemoryMXBean mb;
3     mb = parentJvm.getJmxConnector().getBean(MemoryMXBean.
         class, ManagementFactory.MEMORY_MXBEAN_NAME);
      setUsedMemory(mb.getHeapMemoryUsage().getUsed());
5     setMaxMemory(mb.getHeapMemoryUsage().getMax());
      setInitMemory(mb.getHeapMemoryUsage().getInit());
7     setUsedNonHeapMemory(mb.getNonHeapMemoryUsage().getUsed
         ());
      setMaxNonHeapMemory(mb.getNonHeapMemoryUsage().getMax()
         );
9     setInitNonHeapMemory(mb.getNonHeapMemoryUsage().getInit
         ());
  }
```

Listing 5.7: Memory metrics

As shown is listing 5.7, JVMSencor collects memory statistics from the JMX agent and stores them as attributes.

5.3.3 VI_Monitor

This section gives the main details regarding the implementation of VI_Monitor. It is based on the use of Ganglia, IoStat and MpStat. These tools are either started by VI_Monitor (e.g. MpStat, IoStat) or started during the boot of the machine (e.g. Ganglia). Implementation details regarding the integration of Ganglia, MpStat and IoStat within VI_Monitor are given below.

Ganglia

VI_Monitor typically parses the XML file generated by Ganglia daemons (gmond and gmetad), to get the values of collected metrics and send them via sockets to the analysis agent. Listing 5.8 shows the method used to parse the XML file generated by Ganglia.

```
1 public HashMap<String, String> processXML(String xml) {
```

```
    DocumentBuilderFactory dbf = DocumentBuilderFactory.
       newInstance();
    DocumentBuilder db = null;
    try {
        db = dbf.newDocumentBuilder();
    } catch (ParserConfigurationException e) {
        e.printStackTrace();
    }
    StringReader reader = new StringReader(xml);
    InputSource inputSource = new InputSource(reader);
    Document doc = null;
    try {
        doc = db.parse(inputSource);
    } catch (SAXException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    reader.close();
    doc.getDocumentElement().normalize();
    NodeList nodeLst = doc.getElementsByTagName("METRIC");
    HashMap<String, String> metrics = new HashMap<String,
       String>();
    for(int i = 0 ; i < nodeLst.getLength(); i++) {
      Node n = nodeLst.item(i);
      if (n.getNodeType() == Node.ELEMENT_NODE) {
      }
     NamedNodeMap map = n.getAttributes();
     metrics.put(map.getNamedItem("NAME").toString(), map.
        getNamedItem("VAL").toString());
    }
    nodeLst = doc.getElementsByTagName("HOST");
     return metrics;
}
```

Listing 5.8: Ganglia: Parsing method

SysStat: IoStat and MpStat

VI_Monitor starts IoStat and MpStat, while using the Java code shown in Listing 5.9.

```
Process p=Runtime.getRuntime().exec("iostat -d 15");
```

Listing 5.9: Starting procedure of IoStat

After starting IoStat and MpStat, VI_Monitor stores the outputs in a file, parses the generated file and sends the extracted metrics to the analysis agent, via TCP/IP sockets.

5.3.4 PI_Monitor

PI_Monitor is based on the use of Ganglia, IoStat, MpStat and Xenmon. PI_Monitor uses the same method as VI_Monitor to extract data from Ganglia, IoStat and MpStat. The source code of Xenmon has been modified to send collected metrics to the analysis agent via TCP/IP sockets (see Listing 5.10)

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((CEPIP,port))
s.send("Xenmon,"+str(socket.gethostname())+","+str(cpuidx)
   +","+str(domain_id[dom])+","+str(h1[dom][0][0])+","+str(
   h1[dom][0][1])+","+str(h1[dom][0][2])+","+str(h1[dom
   ][1])+","+str(h1[dom][2][0])+","+str(h1[dom][2][1])+","+
   str(h1[dom][2][2])+","+str(h1[dom][3][0])+","+str(h1[dom
   ][3][1])+","+str(h1[dom][3][2])+","+str(h1[dom][4])+","+
   str(h1[dom][5][0])+","+str(h1[dom][5][1]))
```

Listing 5.10: Modification of Xenmon

PI_Monitor, VI_Monitor, P_Monitor and S_Monitor make use of basic TCP/IP sockets to send collected data to the analysis agent. Implementation details regarding the analysis agent are given below.

5.4 Analysis

The analysis agent of CEP4Cloud is based on the use of the Esper CEP engine to process elementary monitoring events, characterize the state of the Cloud and notify the action manager framework, if a performance-related problem occurs. Our analysis agent is mainly based on the analysis queries of CEP4CMA and the dynamic architecture of D-CEP4CMA. Implementation details regarding CEP4CMA and D-CEP4CMA are given in Sections 5.4.1 and 5.4.2, respectively.

5.4.1 Implementation of CEP4CMA

CEP4CMA implements the proposed analysis rules as EPL queries to process the received monitoring events and characterize the state of the Cloud. Thus, it follows three steps. First, it receives the monitoring data, formats it, sends it and registers it as elementary events within the CEP engine. Second, it implements the proposed analysis rules as EPL queries. Third, CEP4CMA generates complex events, describing the state of the Cloud and registers them within the sinks. So, CEP4CMA relies on the three main packages of CEP4Cloud: sensors, analysis

and sinks. Implementation details regarding these three Java packages are given in the remainder of this section.

The "sensors" package

The "sensors" package allows us to record incoming events, format and send them to the CEP engine. As shown in listing 5.11, received monitoring events are firstly formatted (see Line 1 of Listing 5.11). Afterwards, the resulting events are used to instantiate a new object of the corresponding sensor (see Line 2 of Listing 5.11). The created object is finally sent to the CEP engine.

```
1 formatMSG(MsgR, 4, metricsTexec);
  TexecSensor TS = new TexecSensor(metricsTexec);
3 cepRT.sendEvent(TS);
```

Listing 5.11: Register the incoming data within the CEP sensors

It should be pointed out that the "EPRuntime" class allows us to send recorded events to the CEP sensors. Actually, the object "cepRT" (see listing 5.11) is an instance of the "EPRuntime" class. It is used to register the incoming event with the CEP sensor (TS). The code snippet of Listing 5.11 shows the formatting and sending procedure of the Texec sensor.

Each monitored metric has its own sensor class. The sensor class defines the fields (i.e. attributes) and get methods related to this metric.

The "analysis" package

The "analysis" package mainly includes the implementation of the proposed analysis rules as EPL queries. It allows us to register elementary events within the CEP engine and process them, while using a set of EPL queries. The first step consists of setting the CEP configuration up (see Lines 2,3 and 4 of Listing 5.12) and registering the elementary events (see Line 6 of Listing 5.12).

```
1 //CEP configuration
  Configuration cepConfig = new Configuration();
3 cepConfig.getEngineDefaults().getExecution().
     setDisableLocking(true);
  cepConfig.getEngineDefaults().getThreading().
     setThreadPoolOutbound(true);
5 //Register the received elementary events with the CEP
     engine
  cepConfig.addEventType("BytesInData", BytesInSensor.class.
     getName());
```

Listing 5.12: The first step of the analysis: Configure the CEP and add event types

The second step of the analysis consists of implementing and launching the CEP queries. For instance, listing 5.13 represents an example of an EPL query. The shown query allows us to detect a degradation of the communication time. It is the implementation of the analysis rule (R1) shown in Formula 4.4.

```
select * from JoinTcomB
match_recognize (partition by IP
measures A.IP as id,
A.Tcom as tcom_v1, B.Tcom as tcom_v2,
C.Tcom as tcom_v3, D.Tcom as tcom_v4,
A.PBytesIn as bi_v1, B.PBytesIn as bi_v2,
C.PBytesIn as bi_v3, D.PBytesIn as bi_v4,
A.PBytesOut as bo1, B.PBytesOut as bo2,
C.PBytesOut as bo3, D.PBytesOut as bo4,
A.TimeValue as TVBegin,
D.TimeValue as TVEnd
pattern (A B C D)
define
B as B.Tcom > A.Tcom and ((B.PBytesOut < A.PBytesOut) or (B
   .PBytesIn < A.PBytesIn)),
C as C.Tcom > B.Tcom and ((C.PBytesOut < B.PBytesOut) or (C
   .PBytesIn < B.PBytesIn)),
D as D.Tcom > C.Tcom and ((D.PBytesOut < C.PBytesOut) or (D
   .PBytesIn < C.PBytesIn)))
```

Listing 5.13: An example of a CEP query

As shown in Listing 5.14, the third step of the analysis deals with creating (see Line 1 of Listing 5.14) and registering the EPL queries within the corresponding sink (see Line 2 of Listing 5.14).

```
IOStatement = cepAdm.createEPL(IOrule);
IOStatement.addListener(new IOListener());
```

Listing 5.14: Create and register CEP queries

The last step of the analysis consists of starting the queries (see Listing 5.15).

```
MainReceiver(cepRT);
```

Listing 5.15: Receive data and start queries

The “sinks” package

The “sinks” package is used to subscribe to complex events. It allows us to trigger alarms when a performance-related problem is detected. Listing 5.16 shows an example of a sink that detects an I/O performance-related problem, prints it and saves it in a log file.

```
public class Alarm implements UpdateListener {
    @Override
    public void update(EventBean[] arg0, EventBean[] arg1)
       {
        System.out.println("*******I/O Performance-related
           Problem*******"+arg0[0].getUnderlying());
        Logger.getLogger("FileIO").info("I/O Performance-
           related Problem !! "+arg0[0].getUnderlying());
        }
}
```

Listing 5.16: A sink class

Our analysis agent is based on a dynamic architecture, called D-CEP4CMA. It dynamically switches between centralized and distributed CEP architectures depending on the load/memory of the CEP machine and network traffic conditions in the observed Cloud environment. Implementation details regarding D-CEP4CMA are given below.

5.4.2 Implementation of D-CEP4CMA

D-CEP4CMA is based on two main algorithms (see Figures 4.32 and 4.33). They allow us to dynamically choose the appropriate architecture (centralized or distributed) for the analysis. Both algorithms check the state of the Cloud and the machine hosting the CEP engine to activate the suitable design and its related components for the analysis. The implementation of D-CEP4CMA algorithms is included in the "de.umr.dynamicalgo" package.

To activate architectural components, D-CEP4CMA algorithms connect to the Cloud machines via ssh and remotely launch scripts allowing us to activate / deactivate components. Listing 5.17 shows the connection procedure.

```
JSch jsch=new JSch();
jsch.setKnownHosts("~/.ssh/known_hosts");
Session session=jsch.getSession(user, host, 22);
session.setPassword(passwd);
java.util.Properties config = new java.util.Properties();
config.put("StrictHostKeyChecking", "no");
session.setConfig(config);
```

Listing 5.17: SSH connection

After connecting to the Cloud machine, D-CEP4CMA launches special scripts to activate / deactivate architectural designs. Listing 5.18 gives relevant details regarding the activation / deactivation procedure of a component.

```
Channel channel=session.openChannel("exec");
```

```
((ChannelExec)channel).setCommand("cd "+path+""+Newline+
   script);
```

Listing 5.18: Activation / Deactivation procedure of a component

The outputs of the analysis agent (i.e., diagnosis reports) are used by the action manager framework, to identify and apply the adequate recovery action. Implementation details regarding our action manager framework are given in Section 5.5.

5.5 The Action Manager Framework

The action manager framework is in charge of choosing and applying the suitable recovery action, if a performance-related problem occurs. It is based on the repair algorithm shown in Figure 4.36 and is implemented within the sink. Listing 5.19 shows a code snippet of the sink RepairIO. First, it deduces the name of the over-loaded VM and the IP of its physical host (see Lines 2 and 4 of Listing 5.19). Based on the deduced data, the recovery actions (i.e. recovery scripts) are generated (see Lines 8,9,10,13,14 and 16 of Listing 5.19). Then, the RepairIO sink gets the latest recorded values of the number of I/O requests to the physical disk and the host name of the overloaded physical machine, when the performance problem has been detected (see Lines 22 and 26 of Listing 5.19). These values are used to check the success of the applied action. Finally, the RepairIO sink applies our repair algorithm (see Line 29 of Listing 5.19). The "de.umr.cep.action.manager" includes the implementation of our repair algorithm.

```
//VM hostname (deduced from the query outputs)
String VMname = (String) arg0[0].get("vmname");

//Physical host IP (deduced from the query outputs)
String PhyHostIP = (String) arg0[0].get("physiqueIP");

//Action1: Reconfigure database service
String scriptReconf = "./configureDB.sh";
CentraCEP4CMA.A1.setcommand(scriptReconf);
CentraCEP4CMA.A1.setHost(PhyHostIP);

//Action2: Migrate VM
String scriptMigrate = "HOSTNAME="+namelabel+" ./migrateVM.
   sh";
CentraCEP4CMA.A2.setcommand(scriptMigrate);
//Since we use the migration functionality of OpenStack, we
    launch our script from the controller machine
CentraCEP4CMA.A2.setHost(IPControllerDomU);
```

```
18 ActionManager[] AsI = {CentraCEP4CMA.A1,CentraCEP4CMA.A2};

20 //I/O req value recorded when the I/O performance-related
      problem has been detected by the EPL rule
   //This value is used to check the success of the applied
      action
22 float IOvalueDeg = (Float) arg0[0].get("ioreq3");

24 //Hostname of the physical host
   //used to get the value of I/O req after applying the
      recovery action
26 String hostnameIO = (String) arg0[0].get("PN");

28 //Apply the repair algorithm
   As[0].repairCloud(As, IOvalueBefore, "IO", hostnameIO);
```

Listing 5.19: Code snippet of the sink RepairIO

5.6 Summary

This chapter has presented implementation details of CEP4Cloud and its different components. It has shown the benefits of using Aspect-Oriented-Programming to implement our non-invasive monitoring approach (AOP4CSM). Moreover, this chapter has given relevant code snippets showing the main implementation steps of CEP4CMA. Furthermore, implementation details regarding the dynamic CEP architecture have been presented. They show the activation / deactivation procedure of different architectural designs. This chapter has also provided implementation details of the action manager framework. CEP4Cloud and its components have been evaluated via several experiments. Conducted experiments are presented and discussed in Chapter 6.

"The difference between theory and practice is that in theory, there is no difference between theory and practice."

Richard Moore

6 Experimental Results

6.1 Introduction

To evaluate CEP4Cloud and its main components, several experiments have been conducted using two different testbeds. The performed experiments assess many aspects of each particular component and illustrate the merits of CEP4Cloud. This chapter presents the used testbeds, the conducted experiments and their corresponding setups. It is organized as follows. Our testbeds are described in Section 6.2. Section 6.3 deals with assessing the overhead of using AOP4CSM and the multi-layer monitoring agent. The analysis approaches (CEP4CMA and D-CEP4CMA) are evaluated in Section 6.4. Section 6.5 illustrates the merits of the action manager framework. The evaluation of CEP4Cloud is given in Section 6.6. The last section summarizes this chapter.

Parts of this chapter have already been published in [64–67, 69].

6.2 Testbeds

To illustrate our proposal, we used two different testbeds. The first one has been used to evaluate AOP4CSM. It is a private SaaS Cloud that hosts a medical workflow from the area of *sleep research*. The second testbed has been used to assess our multi-layer monitoring agent, CEP4CMA, D-CEP4CMA, our action manager framework and CEP4Cloud. It is a private Cloud computing environment, based on the open source Cloud software OpenStack and hosts web applications such as web services and the benchmark Day Trader. The used SaaS medical workflow and our private Cloud environment are presented in Sections 6.2.1 and

6.2.2, respectively.

6.2.1 Testbed I: A Medical Workflow as a Service

To evaluate AOP4CSM, several experiments have been conducted in a private Cloud. It consists of three dedicated nodes. The first node is a Core 2 Duo (2.4 GHz) with 4 GB of RAM. It is running under the Mac OS X operating system and hosts the BPEL engine. The two other nodes are running under the Ubuntu operating system and host the workers in an Apache Tomcat servlet container. One of them is a Pentium(R) 4 (3.4 GHz) with 2GB of RAM and the other one is a Pentium Dual Core (1.46 GHz) with 3GB of RAM. The monitored Cloud service is running in the two Tomcat workers. The client invokes the BPEL engine, which in turn calls the service hosted in the Tomcat worker. As a Cloud service, a workflow from the area of *sleep research* that has been developed in cooperation with researchers from the MediGrid (Medical Grid) [98] project of the German Grid initiative (D-Grid) is used. This workflow makes use of a set of open source tools (Physio toolkit [100]) to perform an electrocardiogram analysis and uses the produced results to conduct *apnea detection*. More information about this workflow is available in [49].

6.2.2 Testbed II: The OpenStack Cloud Platform

Figure 6.1 depicts the layered-architecture of the Cloud environment used in our experiments. It is composed of one Cloud controller and a set of compute nodes. The Cloud controller machine has a 64-bit CPU, with 250 GB of disk and 16 GB of RAM. The Ubuntu Server 12.04 TLS is used as the operating system for the physical machines (controller and compute nodes). The number and the characteristics of the used compute nodes depend on the particular experimental scenario. They are precised for each conducted experiment. The Xen Server is the used virtualization technology. The open source Cloud software OpenStack [99] has been deployed in order to upload Cloud images, launch, re-size and migrate instances. The used OpenStack release is precised for each experiment. Based on the Ubuntu Cloud image, we build the used image. It includes the proposed monitoring agent. This image has been uploaded into our OpenStack Cloud platform. It is used to launch instances (i.e. virtual machines). A Java Virtual Machine (JVM) and a web application (web service or Day Trader benchmark) are running on all instances. Our testbed environment has a typical Cloud architecture, since it makes use of OpenStack and is composed of the four principal Cloud layers (physical infrastructure, virtual infrastructure, platform and software layers). The physical infrastructure layer is composed of the physical resources, Xen server and OpenStack. The VM instances constitute the virtual infrastructure layer. The JVM and the used web server are the main components of the platform layer. The software layer consists of our web application. The

used web server and web application depend on the experimental scenario and are precised for each experiment.

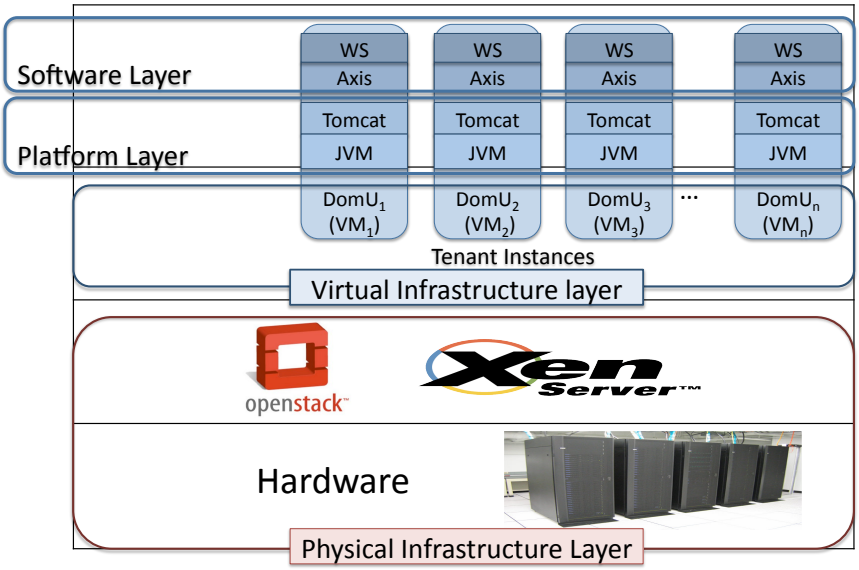

Figure 6.1: The Cloud environment testbed

6.3 Evaluation of the Multi-layer Monitoring Approach

This section presents and discusses the conducted experiments to evaluate AOP4CSM and the multi-layer monitoring agent.

6.3.1 Evaluation of AOP4CSM

To evaluate the performance of AOP4CSM, several experiments have been conducted, using the testbed described in Section 6.2.1. The main objective of these experiments is to evaluate the computational overhead of AOP4CSM. For this purpose, we measured the response time of the used Cloud service (i.e. the medical workflow) while applying the two following scenarios:

- Without AOP4CSM
- With AOP4CSM

Actually, the computational overhead of AOP4CSM is defined as the difference between the response time of a Cloud service that does not use AOP4CSM and the response time of the same Cloud service that uses AOP4CSM. Naturally, these two times are measured without using the functionality of AOP4CSM. The client source code has been modified to compute the response time: timestamps

have been inserted before and after the *call(...)* method. In a first scenario, we executed the medical workflow 100 times without adding AOP4CSM to our platform. The results show that the response time is between 140110 and 140311 milliseconds. The average value is about 140165 milliseconds (see Figure 6.2).

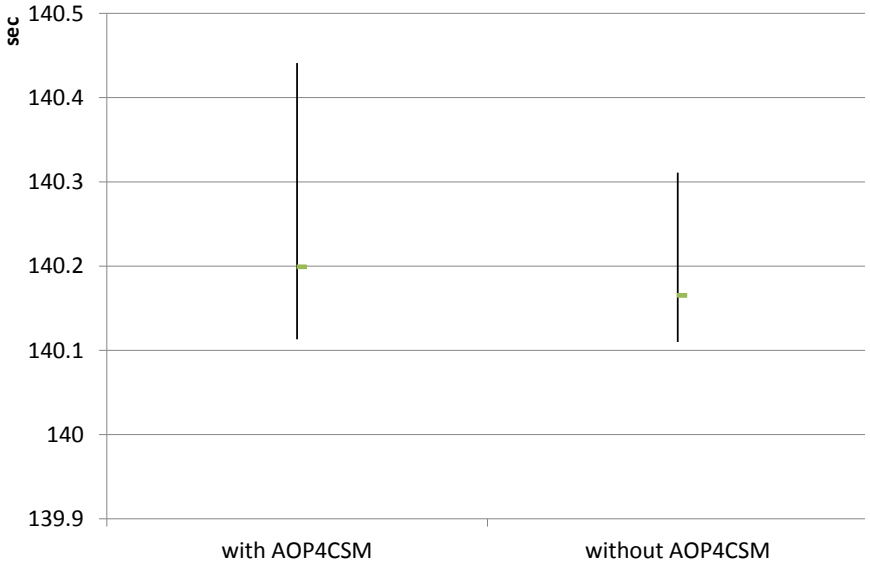

Figure 6.2: Overhead of AOP4CSM

In a second scenario, AOP4CSM (Client and Server components) has been installed on our platform. Again, the same medical workflow has been executed 100 times on the same pool of machines. We also have used the same method (timestamps inside the client source code) to measure the response time. The average value of the response times is about 140199 milliseconds. The values vary between 140113 and 140441 milliseconds (see Figure 6.2).

Thus, the average value of the AOP4CSM overhead is about 34 milliseconds. Its lowest value is around 3 milliseconds, while its highest value is about 130 milliseconds (see Figure 6.2). Thus, the highest value of the overhead is 130 milliseconds, which is considered as a negligible value compared to the response time value (140441 milliseconds). This indicates that the overhead of AOP4CSM is quite low.

6.3.2 Evaluation of the Multi-layer Monitoring Agent

To evaluate the proposed multi-layer monitoring agent, we assessed its CPU overhead. For this purpose, we measured the CPU usage of the physical machine hosting our multi-layer monitoring agent, while varying the number of virtual machines running on this physical machine, and applying the two following scenarios:

- Without the multi-layer monitoring agent
- With the multi-layer monitoring agent

Actually, the CPU overhead of using our multi-layer monitoring agent is defined as the difference between the CPU usage of a machine that does not host the multi-layer monitoring agent and the CPU usage of the same machine that hosts and runs our multi-layer monitoring agent. Naturally, the CPU usage of the physical machine is measured without using the functionality of our multi-layer monitoring agent. The Sar [88] tool has been used to monitor the CPU usage of our compute machine. This experiment has been conducted, using our private Cloud (see Section 6.2.2). In this scenario, we have installed the "Icehouse" release of Openstack and used one compute node. The used compute machine has a 64-bit CPU, with 32 GB of RAM and 250 GB of disk. It hosts 16 virtual machines. Each virtual machine has 1 virtual CPU and 1 GB of RAM, with 10 GB of disk. A Java Virtual Machine (JVM), an Apache Geronimo web server and a Day Trader benchmark [37] are running on all instances. This experiment was running for 20 minutes, 10 times. Figure 6.3 shows the obtained results. It illustrates the average CPU usage of the physical machine, hosting our multi-layer monitoring agent. Experimental results indicate that the CPU overhead of using our multi-layer monitoring agent varies between 0.5% and 1%. Thus, the average value of the CPU overhead of using our multi-layer monitoring agent is equal to 0.8%, which is a negligible value.

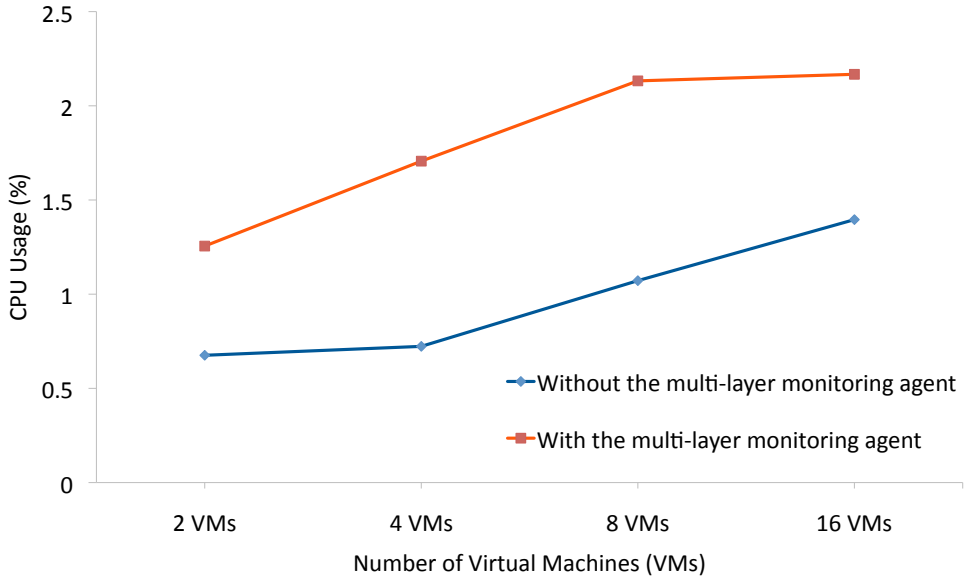

Figure 6.3: CPU overhead of using the multi-layer monitoring agent

6.4 Evaluation of the Analysis Approach

This section presents and discusses the conducted experiments to evaluate our analysis approaches (CEP4CMA and D-CEP4CMA).

6.4.1 Evaluation of CEP4CMA

To evaluate CEP4CMA, two groups of experiments have been conducted. The first group is used to illustrate the benefits of using relationships between metrics for analysis, mainly in terms of rapidity. The second set of experiments allows us to evaluate CEP4CMA, while assessing its precision, recall and diagnosis accuracy.

These experiments have been performed, using the private Cloud described in Section 6.2.2. In this scenario, we installed the "Folsom" release of OpenStack and used 8 compute nodes. Each compute machine has a 64-bit CPU, with 250 GB of disk. Six of them have 16 GB of RAM. The seventh compute node has 12 GB of RAM, and the last one has 8 GB of RAM. For testing purposes and to show the feasibility of CEP4CMA, 80 VM instances have been launched, via OpenStack horizon (i.e. web management interface of OpenStack). Each instance has 1 virtual CPU and 1 GB of RAM, with 10 GB of disk. A Java Virtual Machine (JVM) and an Apache Tomcat web server are running on all instances. Moreover, the Axis engine is deployed on every Tomcat, in order to manage web services. The analysis agent is installed on a dedicated machine, called Cloud Analyzer. It is running under an Ubuntu Server 12.04 TLS. The Cloud Analyzer machine has a 64-bit CPU, with 4 GB of RAM and 140 GB of disk. It receives monitoring data and processes analysis rules to detect performance-related problems.

Rule R_s versus 6 Corresponding Rules

As described in Section 4.4.1, six analysis rules (see Figure 4.25) are replaced by R_s, a very simple rule (see Figure 4.26). The objective of this experiment is to investigate the correctness of this reduction and its benefits with respect to a faster detection of a degradation. First, we compared the quality of the analysis using the 6 rules with the simple rule R_s. For that, we measured the number of triggered alarms of the set of 6 rules and R_s, respectively, while varying N from 2 to 5. N is the number of times the symptoms should be verified to detect a performance-related problem. Since we view a performance degradation as a continuous decrease of the performance parameters, the minimum value of N is equal to 2. However, N also depends on the provider's requirements and the use cases. If the provider requires (for a particular use case) a fast detection process and does not care about false alarms, then 2 is the best value of N. If the provider cares about the number of false alarms (as in our use case), then N should be strictly greater than 2. In this work, 3 has been chosen as the best

value of N, since it eliminates false alarms. This experiment has been executed 5 times, for 30 minutes, while varying the number of nodes. The results show that the number of triggered alarms of the simple rule R_s is almost the same as the number of alarms of the 6 rules.

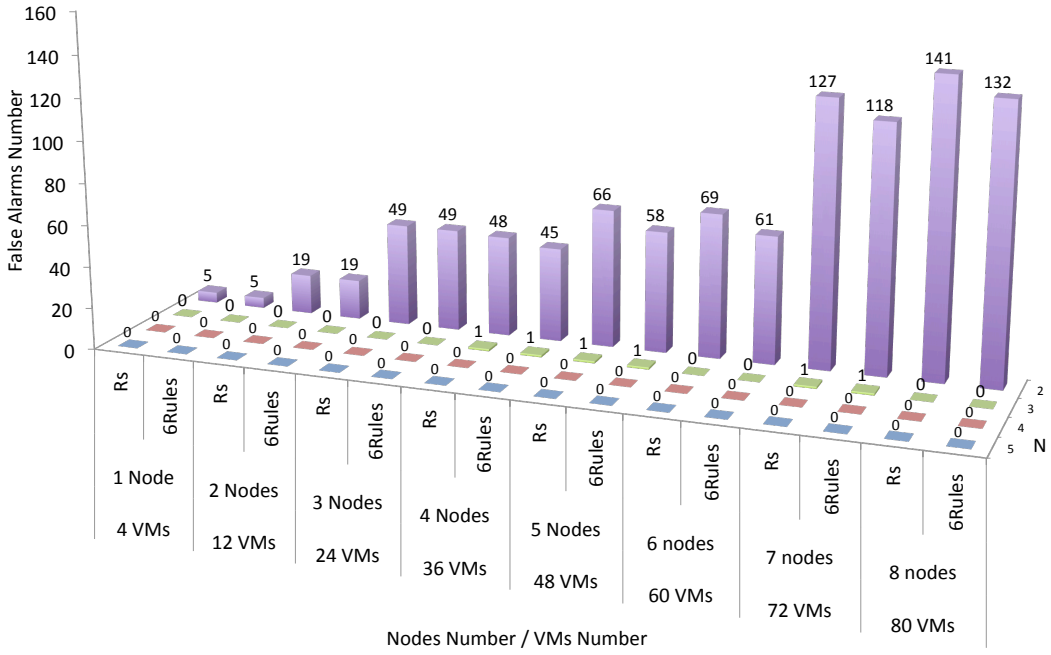

Figure 6.4: Analysis quality: R_s vs. 6 rules

Figure 6.4 depicts the experimental results. It shows that additional alarms are only observed when N=2. For N larger than 2, there are no additional alarms. This illustrates that the quality of analysis offered by R_s is exactly the same as the quality offered by the set of 6 analysis rules, since we will not set N to 2. Thus, the use of the simplified rule allows us to reduce the number of sensors and the number of rules without losing any pertinent information.

Furthermore, using R_s instead of the set of 6 rules allows us to perform a faster analysis. In fact, the next set of experiments shows that the use of R_s is significantly faster than using the set of 6 rules. The experiment consists of injecting a failure (I/O load) via dbench [27] and measuring: (1) the detection time of R_s, and (2) the time needed by the 6 analysis rules to detect the failure. Dbench is a benchmark that generates I/O load. It allows us to launch concurrent clients that perform I/O tasks.

We launched this experiment 5 times, while varying the number of nodes. Figure 6.5 shows the obtained results. It illustrates that R_s takes less time than the six other rules to detect a performance degradation. Figure 6.5 also shows the minimum, average, maximum values of the time needed by R_s and the set of 6 rules to detect a failure. The gain is the difference between the detection

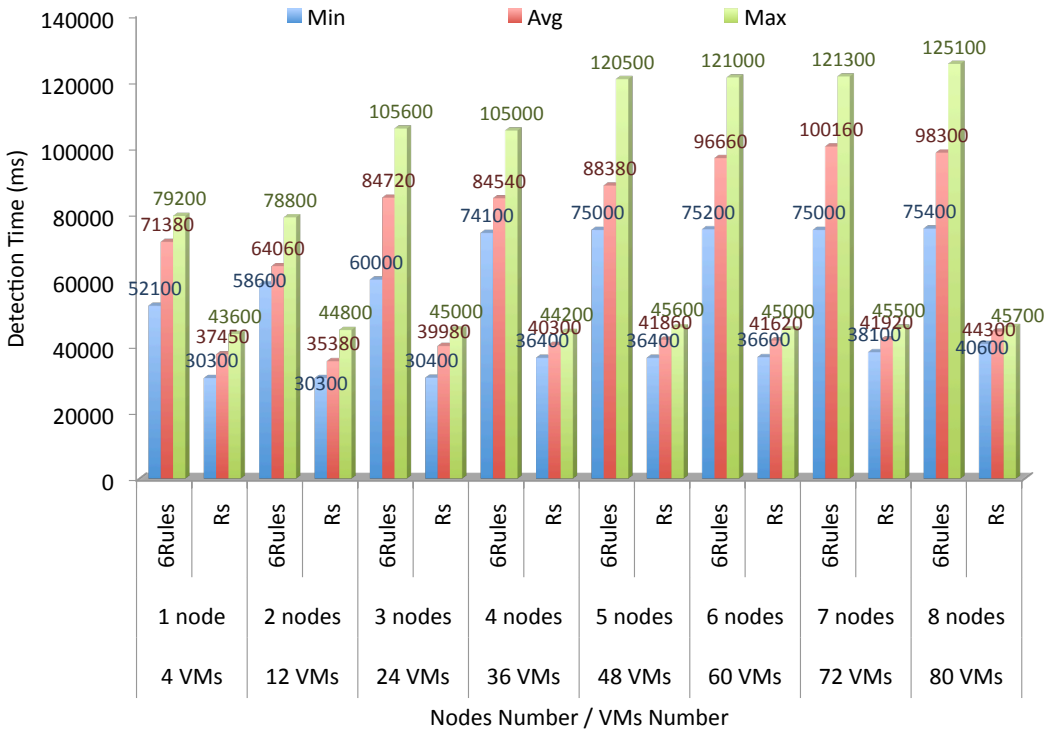

Figure 6.5: Detection time: R_s vs the group of 6 rules

time of the set of 6 rules and the detection time of R_s, divided by the detection time of the set of 6 rules, and multiplied by 100. The gain value varies between 41.8% and 62.8%. Its highest value (62.8%) has been observed in the case of 8 nodes (80 VMs). Thus, the gain is more significant for a large number of nodes. This indicates that our approach is potentially suitable to monitor and analyze a large-scale Cloud computing environment in an efficient manner.

CEP4CMA: Precision/Recall

To evaluate CEP4CMA, we compare it to baseline analysis methods in terms of precision, recall and F1-measure. These three metrics are often used to investigate the quality of failure prediction approaches [87]. The precision expresses the probability of CEP4CMA to generate correct alarms. It is determined by calculating the ratio of the number of true alarms to the number of all alarms (see formula 6.1) [87].

$$Precision = \frac{TP}{TP + FP} \tag{6.1}$$

where TP is the number of True Positive (correct alarms); and FP is the number of False Positive (false alarms).

The recall is defined as the ratio of correct alarms to the number of true failures (see formula 6.2) [87]. It reflects the probability of detecting true failures.

$$Recall = \frac{TP}{TP + FN} \tag{6.2}$$

where FN is the number of False Negative (missing alarms).

The F1-measure was introduced by Rijsbergen [84, 87] to integrate the trade-off between precision and recall. It represents the harmonic (balanced) mean of precision and recall (see formula 6.3) [87].

$$F1 - measure = \frac{2 * Precision * Recall}{Precision + Recall} \tag{6.3}$$

We have chosen threshold-based detection approaches as our baseline methods. We use an 'oracle'-based approach [104] to set a threshold's value. It consists of (1) monitoring the corresponding parameters during a training period, and (2) calculating the lowest and highest 1% of the extracted values, while considering them as outliers. The acceptable range is then taken between the lowest and the highest 1% of the values [104].

We compared CEP4CMA to 3 different threshold-based approaches. The first one is an I/OReq-based method. It checks whether the I/OReq value is in the acceptable range, to decide about the Cloud state. The second one, similar to the first one, is based on the CPU usage metric. The third approach combines the two previous methods. It analyzes the Cloud state according to the values of the I/OReq value and CPU usage.

First, we injected 30 I/O failures, via Dbench, and compared CEP4CMA to the I/OReq-based method. The results of this first experiment show that CEP4CMA is better than the I/OReq-based method, in terms of precision, recall and F1-measure. Actually, CEP4CMA achieves a precision of 89.2%, while the I/OReq-based approach's precision is 57.1%. Moreover, the recall of CEP4CMA is 83.3%, while the I/OReq-based approach achieves a recall of 53.3%. Also, the F1-measure of CEP4CMA (86.2%) is better than the F1-measure of the I/OReq-based method (55.1%). Thus, CEP4CMA outperforms the I/OReq-based method by a precision, recall and F1-measure improvement of 56.2%.

Second, we injected 30 CPU failures, via Sysbench [93], and evaluated the precision and the recall of CEP4CMA and the CPU-based approach. Sysbench is a multi-threaded benchmark tool. It allows us to evaluate operating system parameters, while injecting different kinds of load. Our experimental results show that CEP4CMA achieves better precision (86.6%) than the CPU-based approach (73.9%). Thus, the improvement of precision is around 17.2%. Furthermore, the recall of CEP4CMA is about 86.6%. It outperforms the CPU-based method by an improvement of 58%. In fact, the recall of the CPU-based method is 54.8%. Also, the F1-measure of CEP4CMA (86.6%) is better than the F1-measure of

the CPU-based approach (62.9%). Thus, the improvement of the F1-measure is around 37.6%.

Third, we compared CEP4CMA to the combined approach (based on I/OReq and CPU thresholds). For this purpose, 60 failures have been injected: 50% of them are I/O failures (i.e., injected via Dbench), and the remaining 50% are CPU failures (i.e., injected via Sysbench). In this scenario, we noticed that CEP4CMA achieves a precision of 87.9% and a recall of 85%. Thus, the F1-measure of CEP4CMA is about 86.4%. The precision of the combined approach is about 34.7%, while its recall is around 55%. Thus, the F1-measure of the combined approach is about 42.5%. This means that CEP4CMA is also better than the combined approach: It improves the precision and recall by 153.1% and 54.5%, respectively, and outperforms the F1-measure by an improvement of 103%. Thus, the improvement of the recall is similar for the three conducted experiments. This implies that the three threshold-based approaches have almost the same capabilities in detecting true failures. Using two parameters, in the case of the combined approach, does not make a big difference. Actually, CEP4CMA still achieves similar improvements, when compared to the combined approach. On the other hand, we observed that the combined approach does not reach better results in terms of precision. Indeed, using two threshold comparisons increases the number of false alarms. This is why CEP4CMA outperforms the combined approach by an improvement of 153.1% in terms of precision.

CEP4CMA: Accuracy of the Diagnosis

To demonstrate the accuracy of the diagnosis reports generated by CEP4CMA, we injected 4 different performance-related problems and observed the returned outputs. First, we generated I/O load on the physical machine "compute05", via the benchmark Dbench. In this scenario, we observed that CEP4CMA raised two alarms: the first one indicates that a degradation of the execution time has happened on one of the VMs, belonging to "compute05", due to a high I/O load. The second alarm identified a degradation of the communication time on another VM, also hosted by "compute05". It indicated again that the performance-related problem was caused by a high I/O load. This demonstrates that CEP4CMA was able to correctly identify the overloaded VMs and the degradation's cause. Second, we injected a Java memory failure. To this end, a benchmark that consumes a lot of memory has been implemented. When we started running this benchmark on one of our VMs, CEP4CMA raised an alarm while accurately identifying the overloaded VM and the cause of the degradation. It indicated that the performance problem was caused by an increase of the Java heap memory. Third, we used the Hping3 [43] benchmark to saturate the network links. Hping3 is a networking tool that allows to send TCP/IP packets, and could generate a Denial-of-Service (DoS) attack when used with the flood option. In this scenario, CEP4CMA accurately identified the overloaded physical node. It was also able

to deduce the cause of the performance-related problem: a network congestion. Fourth, we injected a CPU load via Sysbench. CEP4CMA raised an alarm, when Sysbench was running on one of the VMs. It correctly identified the overloaded VM and the degradation's cause (high CPU load).

6.4.2 Evaluation of D-CEP4CMA

To evaluate D-CEP4CMA, several experiments have been conducted, using the private Cloud environment described in Section 6.2.2. In this experiment, we installed the Folsom release of OpenStack and used 4 compute nodes. Each compute machine has a 64-bit CPU, with 250 GB of disk, and 32 GB of RAM. We launched 80 virtual machines (instances), using OpenStack Horizon. Each compute machine hosts 20 virtual machines (VMs). Each VM has 1 virtual CPU and 1 GB of RAM, with 8 GB of disk. A Java Virtual Machine (JVM) and an Apache Tomcat web server are running on all instances. Moreover, the Axis engine is deployed on every Tomcat, in order to manage web services. The CEP Workers are running on the Cloud components (compute nodes and virtual machines). The CEP Manager is running on a separate machine. It has a 64-bit CPU, with 4 GB of RAM and 140 GB of disk. The CEP Manager machine is running under an Ubuntu Server 12.04 TLS (see Figure 6.6).

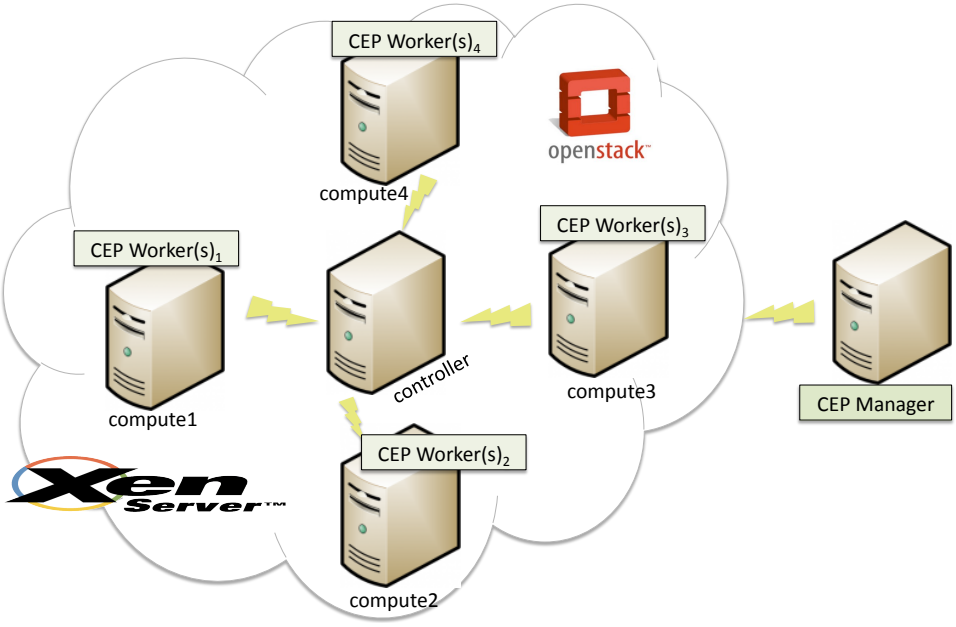

Figure 6.6: Evaluation of D-CEP4CMA: The testbed

The conducted experiments are partitioned into three groups. The first group is concerned with measuring the time (IT) when D-CEP4CMA stops operating to switch between two CEP architectures, called the inactivity time of D-

CEP4CMA. The second group of experiments evaluates D-CEP4CMA in terms of precision and recall. The last group of experiments evaluates the impact of D-CEP4CMA on the machine running the CEP Manager in terms of computational load and used memory.

Inactivity Time of D-CEP4CMA To evaluate the inactivity time, we run D-CEP4CMA for one hour, while varying the number of physical machines composing the Cloud (1..4).

The first experiment was performed with a single Cloud machine (i.e., 20 virtual machines). The centralized CEP architecture was kept until the end of this experiment. Therefore, the inactivity time is equal to 0.

In the second experiment, we added a second machine to the Cloud. We noticed that D-CEP4CMA turned the centralized CEP architecture off after 5 minutes, and replaced it by Design II. The latter was kept until the end of the experiment. The inactivity time was about 7 seconds.

The third experiment was performed with three Cloud machines (i.e., 60 virtual machines). Like before, D-CEP4CMA migrated to Design II after 5 minutes. However, D-CEP4CMA did not keep Design II until the end of the experiment. It switched to Design I 23 minutes later. Then, Design I was kept until the end of this experiment. The inactivity time is the sum of (a) the time needed to switch to Design II and (b) the time spent to migrate to Design I. The inactivity time recorded in this experiment was about 20 seconds.

In the last experiment, our Cloud testbed consisted of four physical machines (i.e., 80 virtual machines). The observed behavior of D-CEP4CMA was similar to the third experiment. D-CEP4CMA migrated to Design II 2 minutes after the start of the experiment. Then, Design II was adopted by D-CEP4CMA for the next 22 minutes. Afterwards, D-CEP4CMA decided to migrate to Design I. The inactivity time was around 22 seconds.

These experiments indicate that the highest value of the inactivity time is about 22 seconds. This value is negligible, in comparison to the duration of the experiment (1 hour).

D-CEP4CMA: Precision/Recall To assess the merits of D-CEP4CMA, we compare it to the centralized architecture, Design I and Design II, in terms of precision and recall.

Our comparison between the different CEP architectures (D-CEP4CMA, Centralized Architecture, Design I and Design II) is based on studying them separately in the same conditions. For each CEP architecture, we (a) injected 30 I/O performance-related problems on the virtual machines, and (b) observed the behavior of each architecture by measuring the number of true alarms, false alarms and missing alarms.

The I/O performance issues were injected via the use of Dbench. The obtained

results are summarized in Table 6.1. It indicates that D-CEP4CMA achieves the same results as the Centralized Architecture in terms of recall and precision. However, it does not suffer from its disadvantages in terms of becoming a bottleneck and a single point of failure.

Table 6.1: Evaluation of CEP architectures

	Centralized Architecture	Design I	Design II	D-CEP4CMA
TP	25	25	23	25
FP	3	2	4	3
FN	5	5	7	5
Precision	89.2%	92.5%	85.1%	89.2%
Recall	83.3%	83.3%	76.6%	83.3%

Impact of D-CEP4CMA on the Machine of the CEP Manager The third group of experiments allows us to compare the performance impact of D-CEP4CMA on the machine running the CEP Manager to the other CEP architectures (centralized architecture, Design I and Design II). For this purpose, we executed the analysis functionality in the different CEP architectures in normal conditions for one hour, and we measured the load and the free memory of the physical machine hosting the CEP Manager.

Figure 6.7 shows that D-CEP4CMA has a negligible impact on the machine running the CEP Manager. The average load of the physical machine hosting

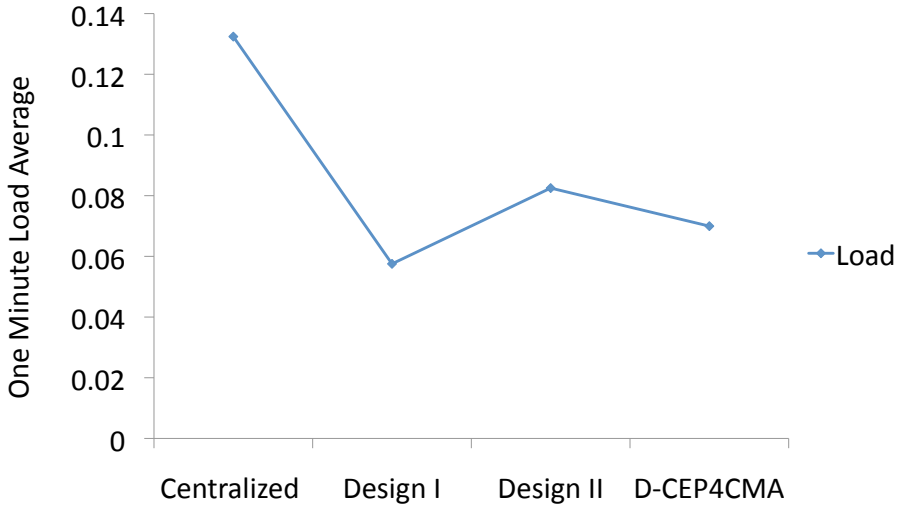

Figure 6.7: Comparison of CEP architectures: CEP manager machine load

the CEP Manager in D-CEP4CMA is lower than the average load measured in the case of the centralized architecture and Design II. However, the average load

of the physical machine hosting the CEP Manager in Design I is slightly lower than the average load measured in the case of D-CEP4CMA. This is related to the D-CEP4CMA strategy. In fact, it uses a centralized CEP architecture during the first stages of its life cycle. This increases the average load of the machine hosting the CEP Manager.

Figure 6.8 indicates that the machine running the CEP Manager consumes less memory in the dynamic CEP architecture (D-CEP4CMA) and Design II. The average free memory of the CEP Manager machine in D-CEP4CMA is lower than the average memory measured in the case of Design II. This is due to the D-CEP4CMA methodology. Actually, it uses a centralized design during its first phases. This leads to a decrease of the average free memory.

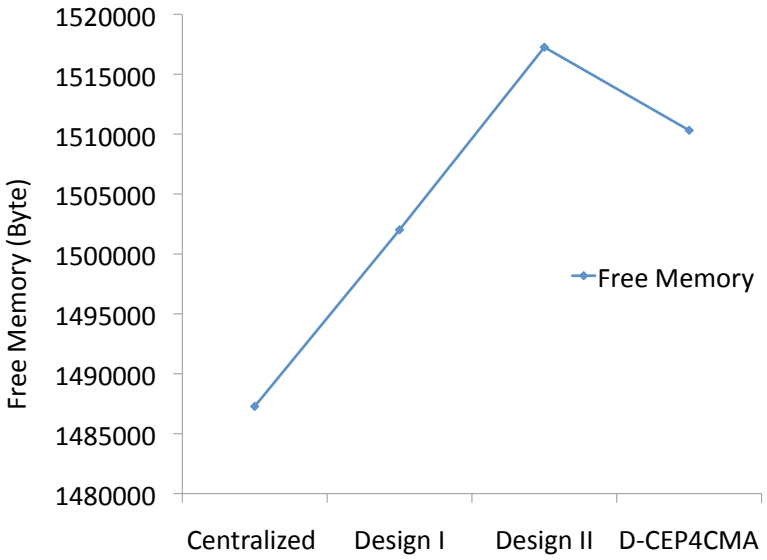

Figure 6.8: Comparison of CEP architectures: CEP manager machine free memory

6.5 Evaluation of the Action Manager Framework

To evaluate the proposed action manager framework, we conducted several experiments, using the private Cloud environment described in Section 6.2.2. In this experiment, we installed the "Icehouse" release of OpenStack and used 5 computes machines. Each physical machine has a 64-bit CPU, with 250 GB of disk. Four of them have 32 GB of RAM and the remaining one has 16 GB of RAM. To show the feasibility of the action manager framework, we launched 80 VM instances. Each instance has 1 virtual CPU and 1 GB of RAM, with 10 GB of disk. A Java Virtual Machine (JVM), an Apache Geronimo web server and a Day Trader benchmark are running on all instances. CEP4Cloud is installed

on a dedicated machine. It is running under an Ubuntu Server 12.04 TLS. The CEP4Cloud machine has an eight-core Intel i7-4771 3.5 GHz processor, 32 GB of RAM and 250 GB of disk. The conducted experiments are partitioned into two groups. The first group is concerned with measuring the overhead of our action manager framework. The second group of experiments allows us to illustrate the benefits of our action manager framework compared to baseline approaches.

6.5.1 Overhead of the Action Manager Framework

To assess the overhead of the proposed action manager framework, we measured the CPU usage and free memory of the machine hosting the action manager framework, while varying the number of machines composing the Cloud, and applying the two following scenarios:

- Without the action manager framework
- With the action manager framework

Actually, the overhead of the action manager framework is defined as the difference between the CPU usage / free memory of a machine that does not host the action manager framework and the CPU usage / free memory of the same machine that hosts and runs the action manager framework. This experiment was running for 20 minutes, 10 times. Figure 6.9 illustrates the average CPU usage of the machine hosting the action manager framework. It indicates that the

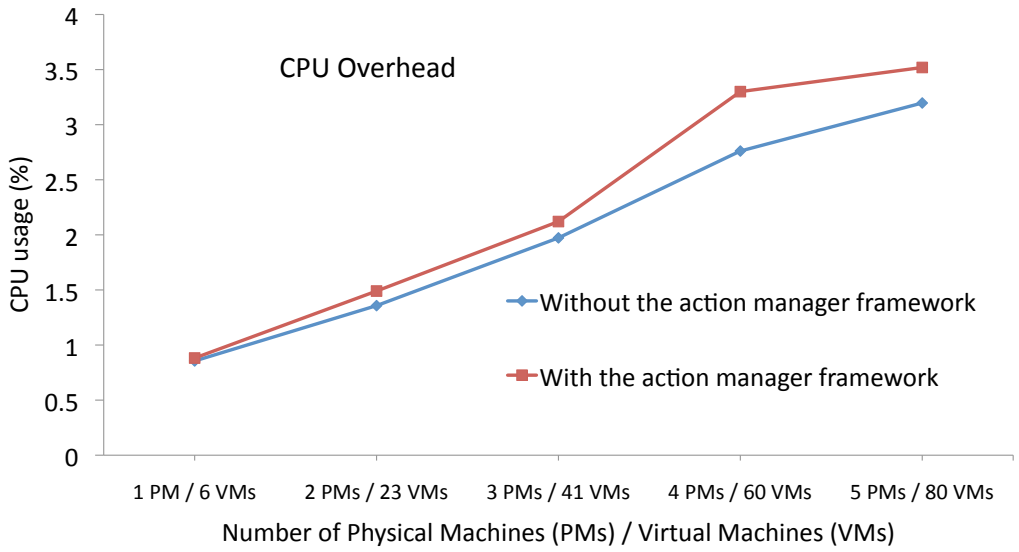

Figure 6.9: CPU overhead of the action manager framework

CPU overhead of our action manager framework varies between 0.02% and 0.5%. Thus, the average value of the CPU overhead of the action manager framework is equal to 0.2%, which is a negligible value.

Figure 6.10 illustrates the average free memory of the machine hosting the action manager framework. It indicates that the RAM overhead of our action manager framework varies between 21 MB and 351 MB. Thus, the average value of the RAM overhead of our action manager framework is equal to 155 MB, which is a negligible value.

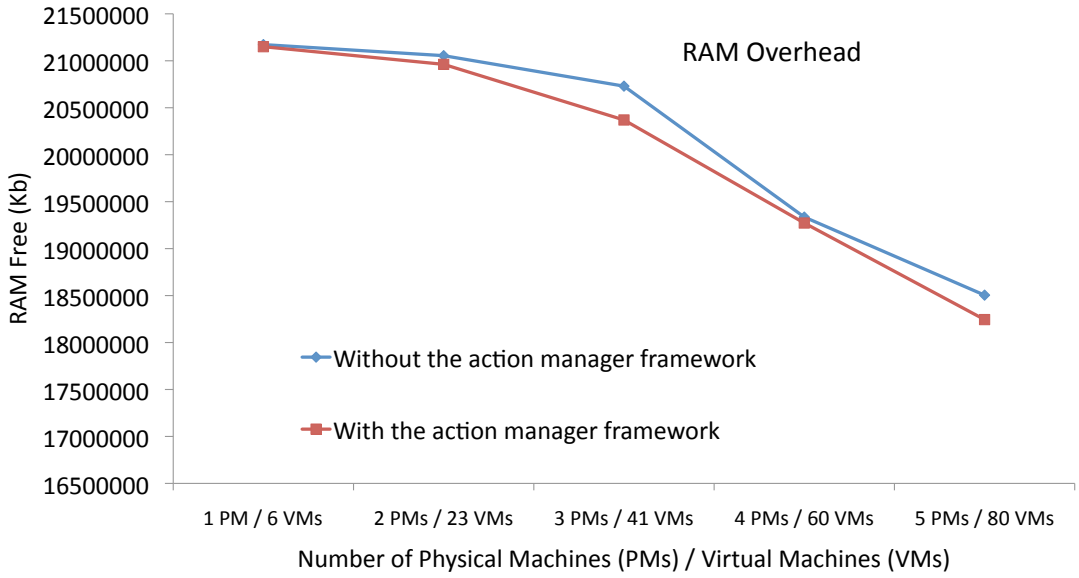

Figure 6.10: RAM overhead of the action manager framework

6.5.2 Action Manager Framework vs. Baseline Approaches

To illustrate the benefits of the proposed action manager framework, we compare it to a baseline approach that triggers an alarm when a performance-related problem occurs. The used baseline approach is called "Alarms-B1". This experiment consists of measuring the number of alarms triggered by our action manager framework (N1), the number of alarms triggered by Alarms-B1 (N2) and comparing N1 and N2. For this purpose, we injected 60 performance-related problems and observed the results returned by our action manager framework and Alarms-B1, respectively. The obtained results indicate that our action manager framework has triggered one alarm (i.e., N1 = 1), while Alarms-B1 has triggered 60 alarms (i.e., N2 = 60). This demonstrates that our action manager framework has reduced the number of alarms by 98.33%.

6.6 Evaluation of CEP4Cloud

To evaluate CEP4Cloud, we used the same testbed setup as in the case of the action manager framework (see Section 6.5). The conducted experiments are

partitioned into three groups. The first group is concerned with measuring the time-to-repair of CEP4Cloud. The time-to-repair (TTR) is the time needed by CEP4Cloud to detect and repair a performance-related problem. It is equal to the sum of the detection time (DT), the time-to-choose (TTC) an action, the time-to-apply (TTA) an action and the validation time (VT), as shown in Formula 6.4.

$$TTR = DT + TTC + TTA + VT \tag{6.4}$$

where:

- DT (Detection Time) is the time needed to detect a performance-related problem.
- TTC (Time-To-Choose an action) is the time spent by our repair algorithm to choose the most adequate recovery action.
- TTA (Time-To-Apply an action) is the time needed to execute the chosen action.
- VT (Validation Time) is the time spent by our repair algorithm to check the success of the applied recovery action.

The second group of experiments allows us to assess the overhead of CEP4Cloud. The third group of experiments is used to illustrate the merits of CEP4Cloud compared to baseline approaches.

6.6.1 Time-to-Repair

To measure the time-to-repair of CEP4Cloud, two experiments have been conducted.

In the first experiment, we injected 20 I/O performance-related problems: 50% of them have been injected via the benchmark sysbench; and the remaining 50% have been injected by launching many virtual machines (>=20VMs) on the same physical node. Our repair algorithm considers two suitable recovery actions to repair an I/O performance-related problem. The first recovery action consists of re-configuring a MySQL database to avoid / fix an I/O bottleneck. The second recovery action allows us to migrate a VM, if many VMs are running on the same physical node. Experimental results indicate that our repair algorithm has always chosen the first recovery action during the first iteration. Actually, the side-effect of the first recovery action (i.e., re-configure MySQL) is less severe than the second one (i.e., migrate a VM). Thus, the first 10 I/O performance-related problems (injected via sysbench) have been repaired by the execution of the first recovery action (i.e., re-configure MySQL). However, the remaining 50% of injected I/O degradations (i.e., many VMs running on the same physical node) have been fixed during the second iteration, via the execution of the second

recovery action (i.e., migrate a VM). Therefore, the average time-to-repair (see Figure 6.11) measured in the case of the first 10 I/O degradations (60.1 seconds) is much more lower than the average time-to-repair observed in the case of the second 10 I/O degradations (805.7 seconds). This is related to the time needed to migrate a VM (733.8 seconds) in the second case (see Figure 6.11).

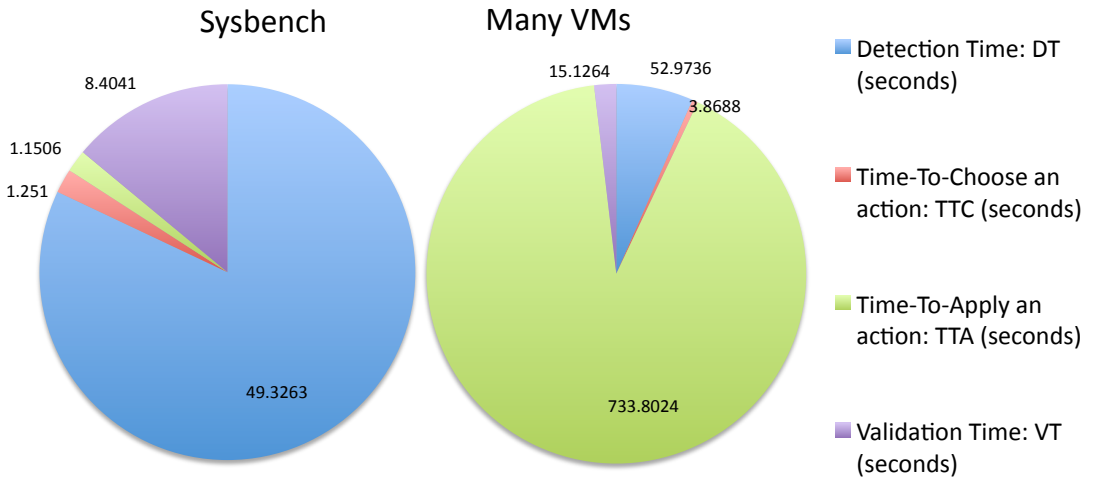

Figure 6.11: Time-To-Repair: I/O performance-related problem

It should be pointed out that we use the VM migration functionality of OpenStack.

The second experiment consists of injecting 10 Java memory failures and measuring the time spent by CEP4Cloud to fix the injected performance-related problem. For this purpose, we implemented a benchmark that consumes a lot

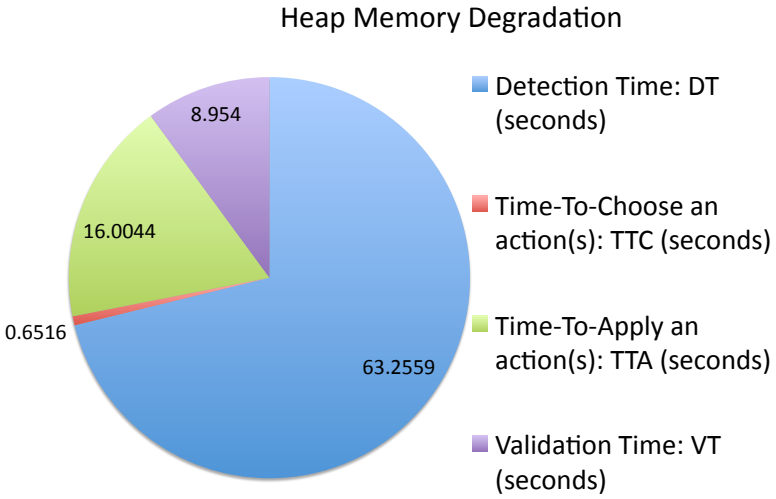

Figure 6.12: Time-To-Repair: High heap memory usage

of Java heap memory. The proposed repair algorithm considers four recovery

actions to fix the problem of high usage of java heap memory. The considered recovery actions allow us to request garbage collection or restart JVM-based applications like the Day Trader application and Apache Geronimo. Depending on the scenario, one of these actions has been executed to repair the problem. Figure 6.12 shows the average time-to-repair spent to fix a java heap memory failure. It is equal to 88.8 seconds: 71% of it is spent on the detection, 11% is spent on choosing and validating the applied recovery action(s) and the rest (18%) is spent on the execution of the recovery action(s). This indicates that our repair algorithm is very fast, since it needs only 9.6 seconds (around 11% of the total time-to-repair) to achieve its tasks.

6.6.2 Overhead of CEP4Cloud

To assess the overhead of CEP4Cloud, we measured the CPU usage and free memory of the machine hosting the CEP engine, while varying the number of machines composing the Cloud, and applying the two following scenarios:

- Without CEP4Cloud
- With CEP4Cloud

Actually, the overhead of CEP4Cloud is defined as the difference between the CPU usage / free memory of a machine that hosts and runs CEP4Cloud and the CPU usage / free memory of the same machine that does not host CEP4Cloud. This experiment was running for 20 minutes, 10 times. Figure 6.13 illustrates the average CPU usage of the CEP4Cloud machine. It indicates that the CPU overhead of CEP4Cloud varies between 0.6% and 3.2%. Its average value is equal to 2%, which is considered to be quite low.

Figure 6.14 illustrates the average free memory of the CEP4Cloud machine. It indicates that the RAM overhead of CEP4Cloud varies between 0 GB and 2 GB. Its average value is equal to 1 GB, which is considered to be reasonably low.

6.6.3 CEP4Cloud vs. Rules-B2

To assess the merits of CEP4Cloud, we compare it to a baseline approach, called Rules-B2, in terms of time-to-repair (TTR).

Rules-B2 makes use of a recovery module that is based on simple "If-Then" rules. Formula (6.5) shows an example of the rules used by the second baseline approach.

$$\begin{aligned} \textbf{if} \ \ & (\text{performance-problem-type} == I/O) \\ & \textbf{Then} \ \ (MigrateVM) \end{aligned} \tag{6.5}$$

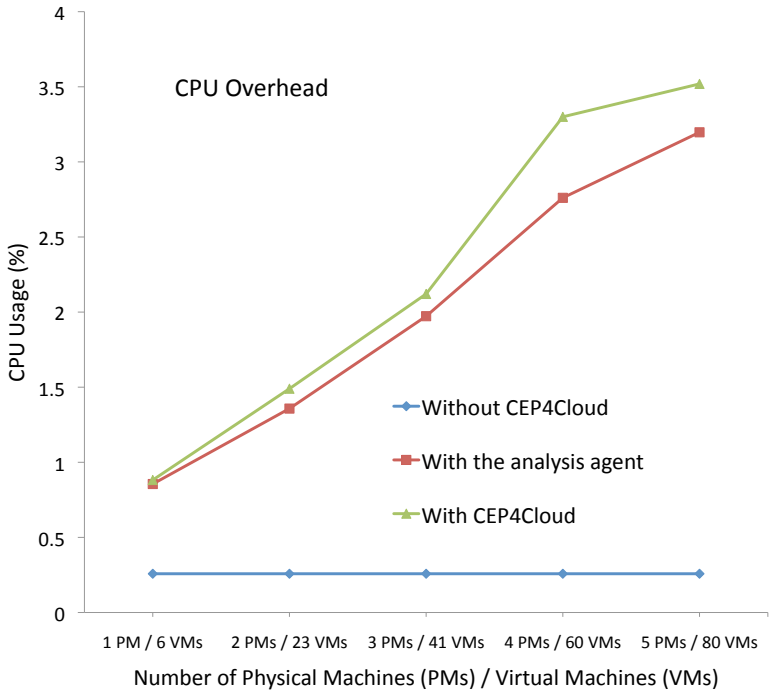

Figure 6.13: CPU overhead of using CEP4Cloud

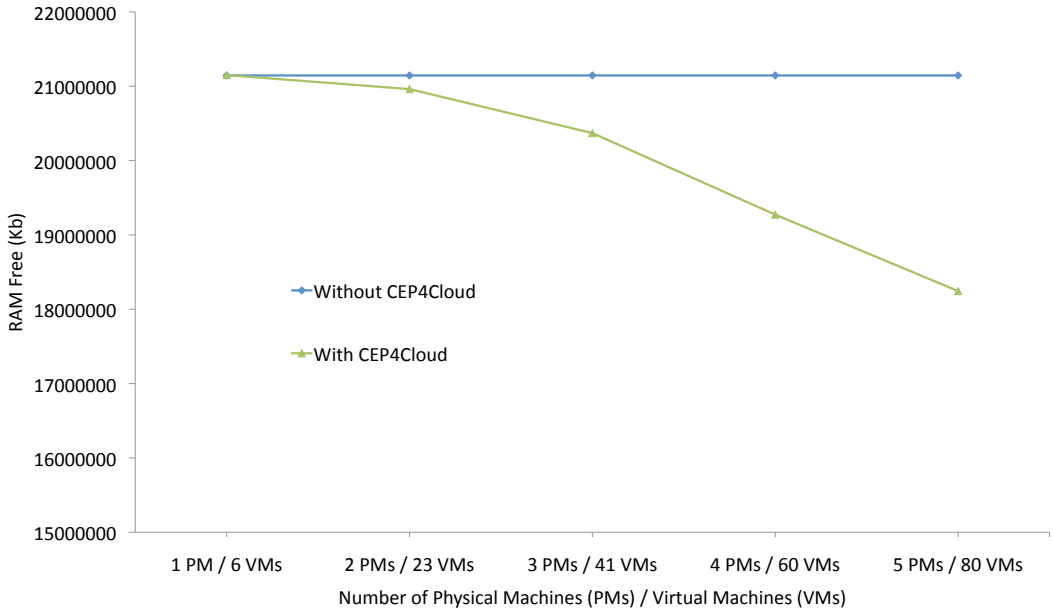

Figure 6.14: RAM overhead of using CEP4Cloud

For this purpose, we injected 10 I/O performance-related problems and measured the TTR of CEP4Cloud and Rules-B2, respectively. 50% of the I/O performance issues have been injected via Sysbench and the remaining 50% are related

to a large number of VMs running on the same physical node. Figure 6.15 illustrates the results. It shows that the average TTR of CEP4Cloud (381.8 seconds) is much lower than the average TTR of Rules-B2 (779.4 seconds). This demonstrates that CEP4Cloud outperforms Rules-B2 by an improvement of 51.01%. This is due to the fact that CEP4Cloud assigns two recovery actions to fix an I/O performance-related problem and chooses the most adequate one. However, Rules-B2 uses the rule shown in Formula (6.5). Thus, it migrates the VM when an I/O performance-related problem occurs. This might increase the TTR.

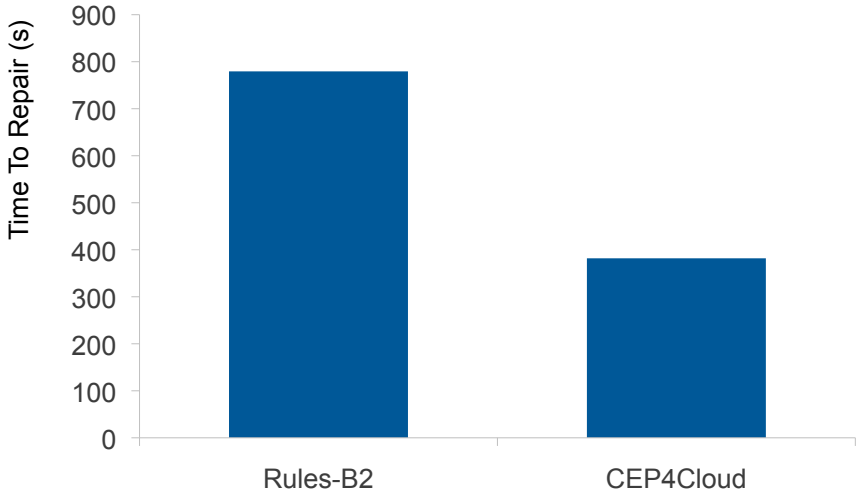

Figure 6.15: CEP4Cloud vs. Rules-B2: Time-To-Repair (TTR)

6.7 Summary

This chapter presented the used testbeds and conducted experiments to evaluate CEP4Cloud and its main components. Our experimental results proved the efficiency and feasibility of CEP4Cloud and its main components. They illustrated that the overhead of using CEP4Cloud and its individual components is low. Moreover, experimental results demonstrated the merits of CEP4CMA compared to threshold-based methods. Furthermore, conducted experiments illustrated the benefits of D-CEP4CMA, in comparison with centralized and distributed designs. Also, the conducted experiments proved the merits of our action manager framework and CEP4Cloud compared to baseline approaches. The next chapter summarizes this thesis and outlines areas for future research.

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Antoine de Saint-Exupéry

7 Conclusion

7.1 Summary

This thesis has presented a cross-layer reactive performance monitoring approach for Cloud computing environments, called CEP4Cloud. It allows us to monitor and analyze performance metrics across Cloud layers, detect performance-related problems and fix them.

CEP4Cloud is based on three new approaches: (1) a multi-layer monitoring approach, (2) a dynamic cross-layer analysis approach and (3) a multi-level recovery approach.

The multi-layer monitoring approach operates at all Cloud layers, while collecting related parameters. It consists of existing tools and a new monitoring approach, called AOP4CSM. AOP4CSM is based on Aspect-Oriented Programming. It monitors quality of service (QoS) parameters of the SaaS layer without modifying the service implementation. AOP4CSM is a non-invasive monitoring approach. Its installation does not need any access to the source code of the service and the client.

The dynamic cross-layer analysis approach is based on using CEP engines to continuously run queries on streams of monitored events across several Cloud layers. It is called D-CEP4CMA for "**D**ynamic **C**omplex **E**vent **P**rocessing for **C**loud **M**onitoring and **A**nalysis". D-CEP4CMA relies on novel analysis rules (i.e., CEP queries) and a new architectural design.

In contrast to other approaches where the queries must be written manually, we derive them from the results of a theoretical and experimental analysis of the relationships of monitored metrics on different Cloud layers and follow a root

cause analysis approach. The proposed approach allows us to reduce the number of sensors and the number of analysis rules (i.e., queries). Furthermore, it does not require a database to store the monitored data, thanks to the use of the CEP engine.

D-CEP4CMA has been designed to dynamically switch between different centralized and distributed CEP architectures depending on the current machine load and network traffic conditions in the observed Cloud environment. It perfectly fits to the elasticity property of Clouds, since it allows us to use the required number of CEP engines when new machines join/leave the Cloud.

The multi-level recovery approach makes use of a novel action manager framework that assigns a set of repair actions to each performance-related problem and checks the success of the applied action. The recovery actions could be applied at all Cloud layers and are updated at runtime.

The conducted experiments have shown the benefits of CEP4Cloud and its components.

First, experimental results have shown that the computational overhead of using AOP4CSM is very low.

Second, obtained results have indicated that monitoring and analysis can be performed very fast without affecting the quality of the diagnosis. They have demonstrated the merits of our analysis approach in terms of precision and recall, in comparison with threshold-based methods. Furthermore, our experimental results have shown that the proposed analysis rules are suitable for the diagnosis of Cloud environments, in the sense that they generate an accurate diagnosis report.

Third, the conducted experiments have shown the merits of the dynamic CEP architecture in comparison to centralized and distributed CEP architectures, in terms of precision and recall. Experiments have also indicated that the time needed to switch between two CEP architectures is negligible.

Fourth, the obtained results have demonstrated that D-CEP4CMA does not overload the machine hosting the CEP engine.

Fifth, the results of several experiments have indicated that the time needed to detect and fix a performance-related problem is reasonably short and that the CPU overhead of using CEP4Cloud is low.

Finally, experimental results have illustrated the merits of CEP4Cloud in terms of speeding up the repair and reducing the number of triggered alarms compared to baseline methods.

7.2 Future Work

There are several areas for future research to be conducted in the field of performance monitoring for Cloud computing environments. Some of them are briefly described in the following.

7.2.1 Dynamic Analysis Rules

The current version of CEP4Cloud is based on static analysis rules that are deduced from our correlation study. The rules and their levels of relevance depend on the specific use case. Indeed, an analysis rule could be triggered many times in the context of use case I and never triggered in the context of use case II. Therefore, it might be interesting to propose new intelligent approaches, based on machine learning techniques, allowing us to dynamically update the used analysis rules at run-time. This includes the following three operations:

- Adding new analysis rules
- Removing unused analysis rules
- Updating analysis rules

7.2.2 Predictive Performance Monitoring

Predictive performance monitoring is a challenging research topic, since it allows to prevent performance-related problems from occurring. Therefore, CEP4Cloud should be extended to predict and prevent performance issues from occurring in Cloud computing environments.

7.2.3 Scalability

Although CEP4Cloud has been well tested on a private OpenStack Cloud computing environment (composed of 80 virtual machines), additional experiments should be performed in a large-scale Cloud environment (i.e., many thousands of machines) to assess the scalability of CEP4Cloud.

7.2.4 Security Intrusions

Since Cloud computing environments suffer from several security issues, it is crucial to extend CEP4Cloud to detect and prevent security intrusions. The idea consists of using the extracted relationships to define new security analysis rules. The defined rules are used to detect security anomalies. For instance, a denial-of-service (DoS) attack can be viewed as a deviation from the normal behavior, when the CPU load is correlated to the number of users requesting the service. Thus, a DoS attack could be detected when the CPU load is increasing and not correlated to the number of users.

7.2.5 Reliability

Ensuring the reliability of Cloud computing environments is a challenging task. CEP4Cloud could be extended to detect and repair failures. This includes propos-

ing new analysis rules allowing us to detect failures and identify their primary causes.

Lists and Registers

List of Figures

List of Tables

List of Listings

Bibliography

[1] Amal Alhosban, Khayyam Hashmi, Zaki Malik, and Brahim Medjahed. Self-Healing Framework for Cloud-Based Services. In *Proceedings of the ACS International Conference on Computer Systems and Applications*, pages 1–7, Fes/Ifrane, Morocco, 2013. IEEE Press.

[2] AOP4CSM. AOP4CSM: Aspect-Oriented Programming for Cloud Service Monitoring. http://www.redcad.org/members/mdhaffar/aop4csm/. Online; accessed 13-November-2014.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communations of the ACM*, 53(4):50–58, 2010.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[5] Axis. Web Services - Axis. http://axis.apache.org/axis/. Online; accessed 13-November-2014.

[6] Bartosz Balis, Bartosz Kowalewski, and Marian Bubak. Leveraging Complex Event Processing for Grid Monitoring. In *Parallel Processing and Applied Mathematics*, volume 6068 of *Lecture Notes in Computer Science*, pages 224–233, Wroclaw, Poland, 2010. Springer.

[7] Luciano Baresi and Sam Guinea. Event-Based Multi-Level Service Monitoring. In *Proceedings of the IEEE 20th International Conference on Web Services*, pages 83–90, Santa Clara Marriott, CA, USA, 2013. IEEE Press.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[9] Riadh Ben-Halima, Khalil Drira, and Mohamed Jmaiel. A QoS-Oriented Reconfigurable Middleware for Self-Healing Web Services. In *Proceedings of the IEEE International Conference on Web Services*, pages 104–111, Beijing, China, 2008. IEEE Press.

[10] Riadh Ben-Halima, Emna Fki, Khalil Drira, and Mohamed Jmaiel. A Large-Scale Monitoring and Measurement Campaign for Web Services-Based Applications. *Concurrency and Computation: Practice and Experience*, 22(10):1207–1222, 2010.

[11] Kanishka Bhaduri, Kamalika Das, and Bryan L. Matthews. Detecting Abnormal Machine Characteristics in Cloud Infrastructures. In *Proceedings of the International Conference on Data Mining Workshops*, pages 137–144, Vancouver, Canada, 2011. IEEE Press.

[12] Subir K. Bhaumik. Root Cause Analysis in Engineering Failures. *Transactions of the Indian Institute of Metals*, 63(2-3):297–299, 2010.

[13] Michael Boniface, Bassem Nasser, Juri Papay, Stephen C. Phillips, Arturo Servin, Xiaoyu Yang, Zlatko Zlatev, Spyridon V.Gogouvitis, Gregory Katsaros, Kleopatra Konstanteli, George Kousiouris, Andreas Menychtas, and Dimosthenis Kyriazis. Platform-as-a-Service Architecture for Real-Time Quality of Service Management in Clouds. In *Proceedings of the 5th International Conference on Internet and Web Applications and Services*, pages 155–160, Barcelona, Spain, 2010. IEEE Press.

[14] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: A High-Performance Event Processing Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1100–1102, Beijing, China, 2007. ACM.

[15] Bu-Qing Cao, Bing Li, and Qi-Ming Xia. A Service-Oriented Qos-Assured and Multi-Agent Cloud Computing Architecture. In *Proceedings of the 1st International Conference on Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 644–649, Beijing, China, 2009. Springer.

[16] Emiliano Casalicchio and Luca Silvestri. Architectures for Autonomic Service Management in Cloud-Based Systems. In *Proceedings of the 16th IEEE Symposium on Computers and Communications*, pages 161–166, Kerkyra (Corfu), Greece, 2011. IEEE Press.

[17] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Open Source Software Development Series, 2008.

[18] N. M. Mosharaf Kabir Chowdhury and Raouf Boutaba. Network Virtualization: State of the Art and Research Challenges. *IEEE Communications Magazine*, 47(7):20–26, 2009.

[19] Chukwa. Chukwa. https://chukwa.apache.org/. Online; accessed 12-November-2014.

[20] Eugene Ciurana. *Developing with Google App Engine*. Apress, Berkely, CA, USA, 2009.

[21] Ira Cohen, Moises Goldszmidt, Terence Kelly, and Julie Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 231–244, San Francisco, CA, 2004.

[22] Collectd. Collectd – The System Statistics Collection Daemon. http://collectd.org/. Online; accessed 11-November-2014.

[23] Douglas C. Crocker. Some Interpretations of the Multiple Correlation Coefficient. *The American Statistician*, 26(2):31–33, 1972.

[24] Gianpaolo Cugola and Alessandro Margara. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys*, 44(3):1–62, 2012.

[25] Yuanshun Dai, Yanping Xiang, and Gewei Zhang. Self-Healing and Hybrid Diagnosis in Cloud Computing. In *Proceedings of the International Conference on Cloud Computing Technology and Science*, volume 5931 of *Lecture Notes in Computer Science*, pages 45–56, Beijing, China, 2009. Springer.

[26] Michal Daszykowski, Krzysztof Kaczmarek, Yvan Vander Heyden, and Beata Walczak. Robust Statistics in Data Analysis – A Review: Basic Concepts. *Chemometrics and Intelligent Laboratory Systems*, 85(2):203 – 219, 2007.

[27] Dbench. Dbench: I/O Benchmark. https://dbench.samba.org/. Online; accessed 13-November-2014.

[28] Shirlei Aparecida De Chaves, Rafael Brundo Uriarte, and Carlos Becker Westphall. Toward an Architecture for Monitoring Private Clouds. *IEEE Communications Magazine*, 49(12):130–137, 2011.

[29] Grzegorz Dyk. Grid Monitoring Based on Complex Event Processing Technologies. Master's thesis, University of Science and Technology in Krakow, 2010.

[30] EdgeRank. EdgeRank. http://edgerank.net/. Online; accessed 12-November-2014.

[31] Franz Faul, Edgar Erdfelder, Axel Buchner, and Albert-Georg Lang. Statistical Power Analyses using G*Power 3.1: Tests for Correlation and Regression Analyses. *Behavior Research Methods*, 41(4):1149–1160, 2009.

[32] FlexiScale. FlexiScale: Utility Computing on Demand. http://www.flexiscale.com/. Online; accessed 13-November-2014.

[33] Ian T. Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Proceedings of the Grid Computing Environments Workshop*, pages 1–10, Austin, TX, USA, 2008. IEEE Press.

[34] Keir A. Fraser, Steven M. Hand, Timothy L. Harris, Ian M. Leslie, and Ian A. Pratt. The Xenoserver Computing Infrastructure: A project overview. Technical Report 552, University of Cambridge, 15 JJ Thomson Avenue Cambridge CB3 0FD, United Kingdom, 2003.

[35] Alan G. Ganek and Thomas A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1):5–18, 2003.

[36] Ganglia. Ganglia Monitoring System. http://ganglia.sourceforge.net/. Online; accessed 12-November-2014.

[37] Apache Geroninmo. Day Trader - Apache Geroninmo J2EE 1.4 Benchmark Sample. http://geronimo.apache.org/GMOxDOC10/day-trader.html. Online; accessed 13-November-2014.

[38] G*Power. G*Power: Statistical Power Analyses for Windows and Mac. http://www.gpower.hhu.de/. Online; accessed 13-November-2014.

[39] Torsten Grabs and Ming Lu. Measuring Performance of Complex Event Processing Systems. In *Proceedings of the 3rd TPC Technology Conference on Topics in Performance Evaluation, Measurement and Characterization*, volume 7144 of *Lecture Notes in Computer Science*, pages 83–96, Seattle, WA, 2012. Springer.

[40] Stephan Grell and Olivier Nano. Experimenting with Complex Event Processing for Large Scale Internet Services Monitoring. In *Proceedings of the 1st International Workshop on Complex Event Processing for the Future*, pages 1 – 10, Vienna, Austria, 2008. CEUR WS series.

[41] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. XenMon: QoS Monitoring and Performance Profiling Tool. Technical Report HPL-2005-187, HP Labs, 2005.

[42] William Von Hagen. *Professional Xen Virtualization*. Wiley Publishing, Inc., 2008.

[43] Hping3. Hping3 - Linux Man Page. http://linux.die.net/man/8/hping3. Online; accessed 13-November-2014.

[44] Markus C. Huebscher and Julie A. McCann. A Survey of Autonomic Computing - Degrees, Models, and Applications. *ACM Computing Surveys*, 40(3):1–28, 2008.

[45] IoStat. IoStat: Linux User's Manual. http://linuxcommand.org/man_pages/iostat1.html. Online; accessed 13-November-2014.

[46] Abhishek Jayswal, Xiang Li, Anand Zanwar, Helen H. Lou, and Yinlun Huang. A Sustainability Root Cause Analysis Methodology and Its Application. *Computers and Chemical Engineering*, 35(12):2786 – 2798, 2011.

[47] Jconsole. The Java Monitoring and Management Console (Jconsole). http://openjdk.java.net/tools/svc/jconsole/. Online; accessed 11-November-2014.

[48] Jpcap. Jpcap – A Network Packet Capture Library for Applications Written in Java. http://jpcap.sourceforge.net/. Online; accessed 11-November-2014.

[49] Ernst Juhnke, Tim Dörnemann, and Bernd Freisleben. Fault-Tolerant BPEL Workflow Execution via Cloud-Aware Recovery Policies. In *Proceedings of 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 31–38, Patras, Greece, 2009. IEEE Press.

[50] Slim Kallel. *Specifying and Monitoring Non-Functional Properties*. PhD thesis, Darmstadt University of Technology, 2011.

[51] Rohit Kamboj and Anoopa Arya. OpenStack: Open Source Cloud Computing IaaS Platform. *International Journal of Advanced Research in Computer Science and Software Engineering*, 4(5):1200–1202, 2014.

[52] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer Journal*, 36(1):41–50, January 2003.

[53] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327 – 353, Budapest, Hungary, 2001. Springer.

[54] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, 1997. Springer.

[55] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: Online Monitoring and Analytics for Managing Large Scale Data Centers. In *Proceedings of the 7th International Conference on Autonomic Computing*, pages 141–150, Washington, DC, USA, 2010. ACM.

[56] Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-Level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm. In *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications*, pages 1–8, Taipei, Taiwan, 2012. IEEE Press.

[57] Long Li, Buyang Cao, and Yuanyuan Liu. A Study on CEP-Based System Status Monitoring in Cloud Computing Systems. In *Proceedings of the 6th International Conference on Information Management, Innovation Management and Industrial Engineering*, pages 300–303, Xi'an, China, 2013. IEEE Press.

[58] Libvirt. Libvirt - The Virtualization API. `http://libvirt.org/`. Online; accessed 13-November-2014.

[59] Amazon Web Services LLC. Amazon Elastic Compute Cloud. `http://aws.amazon.com/`. Online; accessed 13-November-2014.

[60] Joao Paulo Magalhaes and Luis Moura Silva. A Framework for Self-Healing and Self-Adaptation of Cloud-Hosted Web-Based Applications. In *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science*, pages 555–564, Bristol, UK, 2013. IEEE Press.

[61] Meriam Mahjoub, Afef Mdhaffar, Riadh Ben Halima, and Mohamed Jmaiel. A Comparative Study of the Current Cloud Computing Technologies and Offers. In *Proceedings of the 1st International Symposium on Network Cloud Computing and Applications*, pages 131–134, Toulouse, France, 2011. IEEE Press.

[62] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(5-6):817– 840, 2004.

[63] Hidehiko Masuhara and Gregor Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28, Darmstadt, Germany, 2003. Springer.

[64] Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. A Dynamic Complex Event Processing Architecture for Cloud Monitoring and Analysis. In *Proceedings of the IEEE 5th International Conference on Cloud Computing Technology and Science*, pages 270–275, Bristol, UK, 2013. IEEE Press.

[65] Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. CEP4Cloud: Complex Event Processing for Self-Healing Clouds. In *Proceedings of the 23rd IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 62–67, Parma, Italy, 2014. IEEE Press.

[66] Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. CEP4CMA: Multi-Layer Cloud Performance Monitoring and Analysis via Complex Event Processing. In *Proceedings of the 2nd International Conference on Networked Systems*, volume 8593 of *Lecture Notes in Computer Science*, pages 138–152, Marrakech, Morocco, 2014. Springer.

[67] Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. D-CEP4CMA: A Dynamic Architecture for Cloud Performance Monitoring and Analysis via Complex Event Processing. *International Journal of Big Data Intelligence*, 1(1/2):89–102, 2014.

[68] Afef Mdhaffar, Riadh Ben Halima, Mohamed Jmaiel, and Bernd Freisleben. Reactive Performance Monitoring of Cloud Computing Environments. 2014. *Submitted for publication.*

[69] Afef Mdhaffar, Riadh Ben Halima, Ernst Juhnke, Mohamed Jmaiel, and Bernd Freisleben. AOP4CSM: An Aspect-Oriented Programming Approach for Cloud Service Monitoring. In *Proceedings of the 11th IEEE International Conference on Computer and Information Technology*, pages 363–370, Paphos, Cyprus, 2011. IEEE Press.

[70] Afef Mdhaffar, Soumaya Marzouk, Riadh Ben Halima, and Mohamed Jmaiel. A Runtime Performance Analysis for Web Service-Based Applications. In *Proceedings of the 1st Workshop on Engineering SOA and the Web held in conjunction with the 10th International Conference on Web Engineering*, volume 6385 of *Lecture Notes in Computer Science*, pages 313–324, Vienna, Austria, 2010. Springer.

[71] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, 2011.

[72] Haibo Mi, Huaimin Wang, Gang Yin, Hua Cai, Qi Zhou, Tingtao Sun, and Yangfan Zhou. Magnifier: Online Detection of Performance Problems in Large-Scale Cloud Computing Systems. In *Proceedings of the 11th IEEE International Conference on Services Computing*, pages 418 – 425, Washington, DC, 2011. IEEE Press.

[73] MpStat. MpStat: Linux User's Manual. http://www.linuxcommand.org/man_pages/mpstat1.html. Online; accessed 13-November-2014.

[74] Nagios. Nagios is the Industry Standard in IT Infrastructure Monitoring. http://www.nagios.org/. Online; accessed 11-November-2014.

[75] Krishnaprasad Narayanan, Sumit Kumar Bose, and Shrisha Rao. Towards 'Integrated' Monitoring and Management of Data Centers using Complex Event Processing Techniques. In *Proceedings of the 4th Annual ACM Bangalore Conference*, pages 1–5, Bangalore, India, 2011. ACM.

[76] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick H. J. Epema. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *Proceedings of the 1st International Conference on Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, pages 115–131, Beijing, China, 2009. Springer.

[77] Manish Parashar and Salim Hariri. Autonomic Computing: An Overview. In *Proceedings of the International Workshop on Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 257–269, Le Mont Saint Michel, France, 2005. Springer.

[78] John Plummer and Jeff Johnson. Complex Event Processing. Slides, 2008.

[79] Javier Povedano-Molina, Jose M. Lopez-Vega, Juan M. Lopez-Soler, Antonio Corradi, and Luca Foschini. DARGOS: A Highly Adaptable and Scalable Monitoring Architecture for Multi-Tenant Clouds. *Future Generation Computer Systems*, 29(8):2041 – 2056, 2013.

[80] Xen Project. XAPI: Open Source Software to Build Private and Public Clouds. http://www.xenproject.org/developers/teams/xapi.html. Online; accessed 13-November-2014.

[81] Niklas Påhlsson. Aspect-Oriented Programming: An introdution to Aspect-Oriented Programming and AspectJ. pages 1–12. 2002. University Lecture, Departement of Technology, University of Kalmar, Sweden.

[82] Ariel Rabkin and Randy Katz. Chukwa: A System for Reliable Large-Scale Log Collection. In *Proceedings of the 24th International Conference on Large Installation System Administration*, pages 1–15, San Jose, CA, 2010. USENIX Association.

[83] Nicolas Repp, Rainer Berbner, Oliver Heckmann, and Ralf Steinmetz. A Cross-Layer Approach to Performance Monitoring of Web Services. In *Proceedings of the Workshop on Emerging Web Services Technology*, pages 21 – 32, Zurich, Switzerland, 2006. Birkhäuser Basel.

[84] Cornelis J Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 1979.

[85] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services*, pages 205–212. IEEE Press, 2006.

[86] Peter J. Rousseeuw and Mia Hubert. Robust Statistics for Outlier Detection. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):73–79, 2011.

[87] Felix Salfner, Maren Lenk, and Miroslaw Malek. A Survey of Online Failure Prediction Methods. *ACM Computing Surveys*, 42(3):1–42, 2010.

[88] Sar. Sar: Linux User's Manual. http://linuxcommand.org/man_pages/sar1.html. Online; accessed 13-November-2014.

[89] Soumitra Sarkar, Ruchi Mahindru, Rafah A. Hosn, Norbert Vogl, and HariGovind V. Ramasamy. Automated Incident Management for a Platform-as-a-Service Cloud. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, pages 1–6, Berkeley, CA, USA, 2011. USENIX Association.

[90] Peter Sempolinski and Douglas Thain. A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. In *Proceedings of the IEEE 2nd International Conference on Cloud Computing Technology and Science*, pages 417–426, Indianapolis, USA, 2010. IEEE Press.

[91] Jin Shao, Hao Wei, Qianxiang Wang, and Hong Mei. A Runtime Model Based Monitoring Approach for Cloud. In *IEEE 3rd International Conference on Cloud Computing*, pages 313–320, Miami, Florida, USA, 2010. IEEE Press.

[92] BRC Global Standards. Understanding Root Cause Analysis. http://www.tuv-nord.com/cps/rde/xbcr/SID-926CD5F4-935229F0/tng_be_nl/bijlage-nieuwsbrief-januari-2013-brc-understanding-root-cause-an.pdf, 2012. Online; accessed 02-November-2014.

[93] SysBench. SysBench: a System Performance Benchmark. https://launchpad.net/sysbench. Online; accessed 13-November-2014.

[94] Richard Taylor. Interpretation of the Correlation Coefficient: A Basic Review. *Journal of Diagnostic Medical Sonography*, 6(1):35–39, 1990.

[95] TCPDump. TCPDump and LibpCap. http://www.tcpdump.org/. Online; accessed 11-November-2014.

[96] Esper Team. Esper Reference. http://esper.codehaus.org/esper-4.6.0/doc/reference/en-US/pdf/esper_reference.pdf, 2012. Online; accessed 31-October-2014.

[97] GoGrid Team. GoGrid. http://gogrid.com/. Online; accessed 13-November-2014.

[98] MediGrid Team. MediGrid. http://www.medigrid.de/. Online; accessed 13-November-2014.

[99] OpenStack Team. OpenStack: The Open Source Cloud Operating System. http://www.openstack.org/. Online; accessed 02-November-2014.

[100] Physio Team. PhysioToolkit. http://www.physionet.org/physiotools/. Online; accessed 13-November-2014.

[101] Pedro Henriques Dos Santos Teixeira, Ricardo Gomes Clemente, Ronald Andreu Kaiser, and Denis Almeida Vieira-Jr. HOLMES: An Event-Driven Solution to Monitor Data Centers through Continuous Queries and Machine Learning. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*, pages 216–221, Cambridge, UK, 2010. ACM.

[102] Niko Thio and Shanika Karunasekera. Automatic Measurement of a QoS Metric for Web Service Recommendation. In *Proceedings of the Australian Conference on Software Engineering*, pages 202–211, Brisbane, Australia, 2005. IEEE Press.

[103] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[104] Chengwei Wang, Vanish Talwar, Karsten Schwan, and Parthasarathy Ranganathan. Online Detection of Utility Cloud Anomalies using Metric Distributions. In *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium*, pages 96–103, Osaka, Japan, 2010. IEEE Press.

[105] WinPcap. WinPcap: The Industry-Standard Windows Packet Capture Library. http://www.winpcap.org/. Online; accessed 11-November-2014.

[106] Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.

[107] XL. XL - Xen Management Tool, Based on LibXenlight. http://manpages.ubuntu.com/manpages/trusty/man1/xl.1.html. Online; accessed 13-November-2014.

[108] XMind. XMind6 Tool. http://www.xmind.net/. Online; accessed 13-November-2014.

[109] Nezih Yigitbasi, Alexandru Iosup, Dick Epema, and Simon Ostermann. C-Meter: A Framework for Performance Analysis of Computing Clouds. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 472–477, Shanghai, China, 2009. IEEE Press.

[110] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

[111] Qian Zhu, Teresa Tung, and Qing Xie. Automatic Fault Diagnosis in Cloud Infrastructure. In *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science*, pages 467–474, Bristol, UK, 2013. IEEE Press.

Curriculum Vitae

Personal Data

Name	Afef Mdhaffar
Date and Place of Birth	17.10.1983, Sfax, Tunisia
Contact	mdhaffar@mathematik.uni-marburg.de

Education

Jan 2011-	*PhD Candidate,* Joint PhD, Philipps-Universität Marburg, Germany / École Nationale d'Ingénieurs de Sfax, Université de Sfax, Tunisia (ENIS)
July 2009	*Master's Degree,* ENIS, **Grade: Very Good**
Jun 2007	*Diploma in Computer Engineering,* ENIS, **Best Graduation Project**
Juin 2002	*Baccalauréat Mathematics,* Lycée Habib Maazoun, Sfax, Tunisia

Work Experience

2013 - 2014	*Research Assistant,* Philipps-Universität Marburg, Germany
2011 - 2013	*Research Internship,* Philipps-Universität Marburg, Germany, DAAD Grants, Sandwich Model
Jun - Nov 2010	*Research Internship,* Philipps-Universität Marburg, Germany, Project : Service-oriented Distributed Software Architectures
2009 - 2010	*Teaching Assistant,* Faculté des Sciences Économiques et de Gestion de Sfax, Tunisia
2008 - 2009	*Teaching Assistant,* ENIS
2007 - 2008	*Business Intelligence Engineer,* Offshore Decision, Sfax, Tunisia
Feb - Mai 2007	*Engineering Training,* Offshore Decision, Sfax, Tunisia